PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

| | |
|---|---|
| **Titre:** Title: | Debugging and Testing Deep Learning Software Systems |
| **Auteur:** Author: | Houssem Ben Braiek |
| **Date:** | 2022 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Ben Braiek, H. (2022). Debugging and Testing Deep Learning Software Systems [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10496/ |

**Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

| | |
|---|---|
| **URL de PolyPublie:** PolyPublie URL: | https://publications.polymtl.ca/10496/ |
| **Directeurs de recherche:** Advisors: | Foutse Khomh |
| **Programme:** Program: | Génie informatique |

**Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal**
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# Debugging and Testing Deep Learning Software Systems

**HOUSSEM BEN BRAIEK**
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Août 2022

© Houssem Ben Braiek, 2022.

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

**Debugging and Testing Deep Learning Software Systems**

présentée par **Houssem BEN BRAIEK**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

**Giuliano ANTONIOL**, président
**Foutse KHOMH**, membre et directeur de recherche
**Liam PAULL**, membre
**Lionel BRIAND**, membre externe

# DEDICATION

*To my family*

## ACKNOWLEDGEMENTS

Firstly, my deepest gratitude goes to my advisor, Dr. Foutse Khomh, for his guidance, encouragement, and patience during my Master's and Ph.D. studies. Dr. Foutse Khomh is not just a professor today, but also an older brother, who has supported me through my toughest struggles throughout the last five years and celebrated with me both modest and major achievements. Secondly, I also would like to thank Thomas Reid, a senior system engineer at Bombardier Aerospace. During my Ph.D. studies, Thomas Reid introduced me to important background knowledge about aircraft system simulation and modeling processes. He also helped me collaborate and interview aircraft engineers to build applied research tools useful for the aircraft industry. Third, I also want to thank Ali Tfaily and Ciro Guida for their help on my research works during my Ph.D. studies. Likewise, I want to thank all the research students with whom I worked, and especially, Hadhemi Jebnoun, Ahmed Haj Yahmed, and Rached Bouchoucha, who encouraged me, challenged me, and refueled my passion and motivation throughout the journey to my Ph.D. In addition, my thanks go to members of DEEL and its two funding agencies, the National Science and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ). I would also like to thank all DEEL industrial partners, including Thales Canada Inc., Bell Textron Canada Limited, CAE Inc., and especially Bombardier Inc., where I was a resident researcher for more than 2 years during my Ph.D. studies. Furthermore, I would like to thank my committee members, Dr. Guliano Antoniol, Dr. Lionel Briand, and Dr. Liam Paull for evaluating my thesis. Finally, I must thank my dearest family for their great support all the way along my postgraduate studies.

# RÉSUMÉ

L'apprentissage profond est de plus en plus déployé dans les systèmes à grande échelle et les systèmes critiques grâce aux percées récentes en apprentissage automatique. On utilise maintenant des applications logicielles intégrant l'apprentissage profond dans la vie de tous les jours, incluant les finances, l'énergie, la santé et les transports. L'approche traditionnelle du développement logiciel est déductive, consistant en l'écriture de règles qui dictent le comportement du système avec un programme codé. En apprentissage profond, ces règles sont plutôt inférées de façon inductive à partir de données d'apprentissage. Malgré ce changement de paradigme permettant des progrès transformationnels dans plusieurs domaines d'applications, des inquiétudes quant à leur fiabilité ont été soulevées et ont rendu la recherche sur la qualité des logiciels incluant des réseaux de neurones d'une importance primordiale. Premièrement, les programmes d'entraînement des réseaux de neurones modernes sont devenus plus complexes, incorporant le non-déterminisme inhérent et les subtilités cachées derrière une pile de boîtes à outils. En conséquence, les bogues tels que des erreurs de codage, des mauvaises configurations et l'abus d'interface de programmation (API) se produisent fréquemment, selon des études récentes. Par conséquent, nous concevons des débogueurs automatisés qui s'appuient sur l'analyse de code statique et de la dynamique d'entraînement, tout en étant conceptuellement découplés des APIs, des modèles et d'autres choix d'implémentation. Nos outils de débogage sont implémentés pour fonctionner avec les bibliothèques populaires, et leurs évaluations sont menées sur de vrais programmes bogués de StackOverflow et Github. Deuxièmement, même un programme d'entraînement sans bogue peut aussi produire des modèles non fiables en raison de la tendance de l'apprentissage basé sur les données au biais de sélection, aux fausses corrélations et aux déviations de distribution. Ces modèles non fiables se basent sur des raccourcis injustifiés et des biais inductifs médiocres qui ne correspondent pas aux exigences de l'application ou aux propriétés du système. De plus, les développeurs ont tendance à inclure des opérations post-entraînement comme la compression de taille des paramètres, qui peuvent altérer négativement les biais inductifs sans être identifiés avant le déploiement. Traditionnellement, le test du modèle repose sur l'estimation de ses performances sur des échantillons de test qui sont indépendants et distribués de manière identique. Cependant, ces tests échouent souvent à exposer les défauts inhérents du modèle car les échantillons retenus pour le test héritent souvent des mêmes biais que les données d'apprentissage. En outre, les attaques contradictoires se concentrent sur la révélation des vulnérabilités de modèle contre des entrées malveillantes qui ne correspondent pas forcément aux conditions réelles. En parallèle, les tendances de test automatisé

des réseaux de neurones s'oriente vers la conception de critères de couverture structurelle sans preuve de corrélation avec l'aptitude de révélation des défauts, et elles sont évaluées sur leur capacité à guider la génération des images déformées pour les applications de vision par ordinateur. En ce qui concerne les dérives distributionnelles, la modélisation générative est limitée par la malédiction de la dimensionnalité. Les détecteurs dépendants du modèle se basent sur la quantification de l'incertitude ou l'analyse des patrons d'activation, ce qui les rendent sujets à la dégradation de fiabilité face aux entrées significativement décalées. Dans cette thèse, nous construisons des méthodes de test pour les réseaux de neurones, qui sont sensibles au domaine et qui intègrent des métaheuristiques de recherche pour faire face aux larges espaces de recherche. Ces méthodes produisent systématiquement des entrées de test qui évaluent la robustesse nécessaire pour l'application, la cohérence avec les sensibilités déduites de la physique et les inefficacités de la quantification des paramètres. Ces objectifs de test vérifient que les biais inductifs encodés par le modèle ou son homologue quantifié sont alignés avec les exigences de l'application et les processus physiques liés au système modélisé. Continuant l'exploitation des connaissances du domaine, nous développons une nouvelle stratégie de détection des entrées hors distribution qui se base sur la régularité, qui est l'une des propriétés a priori courantes du système, afin de détecter les profils de sensibilité suspects indiquant des entrées assez décalées et probablement hors distribution. Les évaluations de nos méthodes sont menées soit sur des cas d'étude dans divers domaines tels que la reconnaissance d'objets, de la parole et de texte, soit sur des applications d'ingénierie aéronautique, telles que l'optimisation de la conception basée sur des modèles de substitut et la modélisation des performances du système, fournies par notre partenaire industriel, Bombardier Aéronautique.

# ABSTRACT

Deep Learning (DL) is increasingly becoming an integral part of intelligent software systems in critical aspects of our daily lives, from finance and energy to health and transportation. Traditionally, software logic is developed deductively, by encoding the rules that govern the program's behavior into the code. In contrast, Deep Neural Networks (DNNs) derived inductively (i.e., learned) these rules from the training data. Despite this paradigm shift enabling transformational progress across a wide range of domains, concerns over their trustworthiness have been raised and have made research on DL software quality of paramount importance. First, the training programs of modern DNNs have become more complex, incorporating inherent nondeterminism and hidden intricacies of a deeply-growing pile of toolkits. As a result, bugs such as coding faults, misconfigurations, and API misuses occur more frequently, according to recent studies on DL bugs. Therefore, we design automated DL debuggers that rely on static code analysis and training dynamics monitoring, while being conceptually decoupled from APIs, Models, and other implementation choices. Our debugging tools are implemented to work on mainstream DL libraries, and their assessments are conducted on real buggy programs from StackOverflow and Github. Second, even a bug-free DL training program can still produce unreliable DNNs because of the data-driven learning's proneness to selection bias, spurious correlations, and distributional shifts. In such cases, an optimized DNN might have learned unjustified shortcuts and poor inductive biases that misaligned with the application requirements or the system properties. Furthermore, modern DL pipelines tend to include post-training operations such as size compression or domain transition, which can adversely alter the patterns captured by the DNN without being identified before deployment. Conventional testing of a DNN reposes in estimating its performance on independent and identically distributed (iid) held-out test samples. However, iid tests often fail to expose the inherent model's flaws because the held-out samples often inherit the same biases of the training data. Besides, adversarial attacks focus on maliciously-crafted inputs that reveal model vulnerabilities against likely-unforeseeable data in real-world conditions. In parallel, DL software testing trends focus on designing structural coverage criteria with no evidence of correlation to fault-revealing abilities, and they are evaluated on how well they generate distorted images for computer vision applications. In regards to distributional drifts, generative modeling is limited by the curse of dimensionality, while model-dependent detectors based on uncertainties or activation maps suffer from reliability degeneration on shifted inputs. In this thesis, we build search-based and domain-aware DL testing methods that incorporate metaheuristics to cope with high-dimensional search spaces, as well as means to interface with the

domain knowledge. The proposed methods systematically produce test inputs that probe for application-specific robustness, consistency with physics-grounded sensitivities, and elusive quantization inefficiencies. These test objectives ensure that the inductive biases encoded by the DNN or its quantized counterpart are aligned with the application requirements and system-related physics processes. In keeping with domain awareness, we develop a novel out-of-distribution (OOD) detection strategy based on smoothness, which is one of the common apriori system properties, to detect suspicious DNN local-sensitivity profiles that indicate shifted and likely OOD inputs. The evaluations of our methods are conducted on either open-source DL study cases across various domains like object, speech, and text recognition, or aircraft engineering applications, such as surrogate-based design optimization and system performance modeling, provided by our industrial partner, Bombardier Aerospace.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| AI | Artificial Intelligence |
| ML | Machine Learning |
| DL | Deep Learning |
| DNN | Deep Neural Network |
| FNN | Feedforward Neural Network |
| CNN | Convolutional Neural Network |
| RNN | Recurrent Neural Network |
| GP | Gaussian Process |
| RBF | Radial Basis Function |
| ID | In-Distribution |
| OOD | Out-Of-Distribution |
| AX | Adversarial Example |
| CI | Confidence Interval |
| JSD | Jensen–Shannon Divergence |
| ERM | Empirical Risk Minimization |
| SQA | Software Quality Assurance |
| SBST | Search-Based Software Testing |
| CGF | Coverage-Guided Fuzzing |
| MT | Metamorphic Testing |
| MR | Metamorphic Relation |
| PBT | Property-Based Testing |
| GTS | Graph Transformation System |
| GA | Genetic Algorithm |
| PSO | Particle Swarm Optimization |
| FFA | Firefly Algorithm |
| GWO | Gray Wolf Optimizer |
| MFO | Moth Flame Optimizer |
| WOA | Whale Optimization Algorithm |
| MVO | Multi-Verse Optimizer |
| SSA | Salp Swarm Algorithm |
| A/C | Aircraft |
| MDO | Multidisciplinary Design Optimization |
| WAIS | Wing Anti-Icing System |

| | |
|---|---|
| MTOW | Maximum TakeOff Weight |
| TTC | Time To Climb |
| BFL | Balanced Field Length |
| TAT | Total Air Temperature |
| TAS | True Air Speed |
| ALT | Altitude |
| PWR | Pressure at Wing Root |
| TWR | Temperature at Wing Root |
| AOA | Angle of Attack |

# CHAPTER 1 INTRODUCTION

With the continuous innovation in Deep Learning (DL), we have witnessed impressive development of DL software systems; breakthroughs continue to occur in every corner of this field. These intelligent applications have become a part of modern life: security authentication, e-health, and driverless transportation. By 2030, AI is expected to produce extra economic profits of 13 trillion U.S. dollars, which contribute 1.2% annual growth to the GDP of the whole world [2]. However, along with their rapid and revolutionary advances, DL software systems have also exposed their untrustworthy sides through the widely-reported fatal accidents of self-driving cars [3, 4]. Thus, concerns around trustworthiness have become a huge obstacle for DL to overcome to further advance as a field, and become more widely adopted. Therefore, research on quality assurance(QA) methods and tools that can aid in releasing trustworthy DL software systems is now a high priority for academia and industry. In the following, we introduce the DL software systems, their trustworthiness challenges, and the limitations of existing QA approaches. Then, we present our research statement, the outline and the contributions of the present thesis.

## 1.1 Deep Learning Software Systems

Deep Learning (DL) modernizes various legacy neural network ingredients to enhance the capacity of learning and remove the preliminary feature engineering step. DL stacks thousands of computational layers and makes them learning through the leverage of normalization layers [5], residual shortcut connections [6], and momentum-based stochastic optimizers [7]. The data-to-model pipeline remains fairly consistent with similar main steps: data ingestion, preprocessing, model building, training, and performance evaluation. In addition, DL introduces advanced operations over classic machine learning to support designing system models for different application use cases beyond standalone explanatory or predictive statistical models. Among the operations, transfer learning allows the reuse of pretrained models on new applications, whereas post-training compression methods, such as quantization or pruning, allow the reduction of the model footprint for deployment on edge devices. Nowadays, DL software systems represent the next generation of software systems [8] that comprise DL components besides traditional software components. DL components include (1) the data that are collected and preprocessed; (2) the deep neural network (DNN) that is specified and designed for solving a targeted learning problem; (3) the training program that implements the iterative learning algorithm to fit the DNN's parameters on the data. DL components

equip a software system with self-learning capabilities through optimized DNNs [9] that statistically derive their logic from the collected data. By inferring the logic of the system based on data, the lack of or difficulty of encoding formally the specification is mitigated. For instance, DL components can be an inexpensive alternative to building aircraft system performance models without thoroughly formulating the physics-based differential equations governing the system's behavior.

### 1.2 Disruptive Challenges to Trustworthy DL systems

Firstly, the training program is the most software-intensive component in the DL pipeline, which can be buggy like any traditional program, as reported by previous studies [1, 10, 11]. Since the training program implements the DNN design and its learning algorithm, a bug may obstruct the DNN optimization or worse, prevent the training process from converging to optimal DNNs without explicit errors [12]. Listed below are three of the most common root causes of DL bugs that we address in this thesis.

**Coding Faults:** Like any human written code, DNN training programs may contain missing and wrong code statements that cause a deviation between the algorithm and its implementation. For instance, most of these coding faults result in incorrect math operations, leading to erroneous outcomes such as flipped sign result, inverted order of calculations, or wrong data transformations.

**Misconfigurations:** DNN training programs implement highly-configurable algorithms, where setting up the appropriate configuration, given the context, becomes a challenging task. Hence, misconfigurations assemble all the wrong and poor choices for the configuration of the training program's components, including the DNN design and the optimization routines. For instance, most misconfigurations in relation with random initializers, loss functions, normalization methods and optimization hyperparameters, lead to training pathologies and likely inefficient DNNs.

**API Misuses:** While DL libraries provide a rich set of API routines, their generic-purpose nature imposes many assumptions that practitioners should be aware of. As an example, most of these routines' arguments are set by default to recommended values that may not be appropriate to solve certain problems. Thus, API misuses, i.e., the usage of API routines without fully understanding their inner functioning and–or checking that their assumptions are fulfilled, are likely to yield unexpected results such as runtime exceptions or ineffective learning.

Secondly, a bug-free and correctly-designed DNN training program is able to produce an optimized DNN that fits the training samples, i.e., yielding low loss values. Nevertheless, there is no guarantee that this best-fitted DNN has captured useful patterns and essential structure to perform as expected in deployment scenarios. Indeed, the DNN parameters fitting that is primarily guided by the average loss minimization on training data is prone to minority ignorance, confounding factors, and overfitting. In addition, modern DL pipelines can include post-training operations on parameters like quantization, which should be carefully performed to avoid catastrophic degradation. Unfortunately, the conventional testing that evaluates the DNN performance on independent and identically distributed (iid) data samples that should be held-out to be a proxy for future entries, is agnostic to these misconceptions. Indeed, the test sample is generally a portion of the collected data similarly to its training counterpart; so it is likely to inherit the same biases. In the following, we detail the DNN misconceptions associated with the above-mentioned issues that are tackled by our research works.

**Weak Inductive Biases:** During the optimization, the DNN encapsulated weak inductive biases that cannot enable generalization beyond the training distribution because of spurious correlations, i.e., features that are strongly associated with the label in the training data, but are not associated with the label in some practically important settings. Indeed, DNN fitting using empirical risk minimization is prone to shortcut learning, where the DNN learns easily how to generalize well on the collected iid data, but only some of its patterns are well-aligned to the intended solution to the prediction problem.

**Risky Post-Training Compression:** When performing a post-training operation on the DNN parameters, practitioners simply watch for potential degradations of predictive performance on test samples. However, the induced variations in the DNN's learned patterns that are practically relevant for the prediction problem, remain uncovered during evaluations on arbitrary test examples, while adversely impacting its robustness and generalizability in real-world scenarios.

**Acute Distributional Shifts:** The optimal DNN based on iid performance evaluations may fail dramatically in operation because of structural mismatches between training and deployment; thus, the inductive biases encoded by the DNN become less credible. In fact, if iid data suffers from selection bias or out-of-dateness, the optimized DNNs are limited by their DL pipeline design, and hence, they should be avoided under operating conditions that are substantially different from those exploited by the DL pipeline.

### 1.3 Limitations of Current DL Software Quality Assurance Trends

Adversarial attacks [13] focus on DNN robustness assessment against constrained input perturbations that are maliciously crafted to falsify the DNN predictions. They facilitate early detection of security vulnerabilities aiming to reinforce the defense mechanisms. Besides, structural coverage-guided DL testing approaches [14] use input perturbations along with designed data transformations to enrich the test data and derive synthetic test inputs that are able to trigger uncovered structural patterns of DNN activations. Despite their success in exposing DNN failures, subsequent studies [15] show that uncovering these activation patterns is not directly correlated to enhancing the rate at which DNN faults are exposed, but it indeed indicates certain induced-behavioral diversity among the generated test inputs. In regards to the exposed failures, they result from sampling repeatedly from the large input space of perturbed/transformed data. Aside from structural coverage, these existing DL testing methods are primarily designed to explore the derived synthetic test inputs within the neighbors of original sets, which limits their application to certain data transformations [16]. In consequence, most of their evaluations are exclusively conducted on computer-vision models, where the input images are simply floating-point 3D arrays [17–23]. Furthermore, DL testing trends continue to focus on the development of innovative testing components, such as adequacy criteria and test generation algorithms, while maintaining straightforward data derivation rules and baseline DNNs as proof-of-concept implementations. Nonetheless, the disconnection between application requirements and actual DL test cases reduces the generation of DL test cases into simply a uniform degradation of DNN performance. The latter occurs because the employed data transformations are capable of producing synthetic inputs sufficiently different from their originals, but the test generation process has no application-sensitive guidance to assess the credibility of the DNN's inductive biases in regards to relevant dimensions that have to be handled by a candidate solution to the underlying prediction problem. In regards to distributional drifts, the curse of high dimensionality limits the use of generative models for recognizing the out-of-distribution (OOD) data points within the input space, while current trends in model-dependent OOD detection involve inferring uncertainty estimates along with prediction or exposing OOD inputs based on unexpected patterns of activation. Clearly, these techniques rely on either a data-driven calibration or data-related features that may degrade in reliability with significant distributional shifts and can be misleading as well.

## 1.4 Research Statement

Investigating DL bug studies and former quality assurance methods, we made the following observations:

- DL crash-inducing bugs are typically caused by coding errors or API misuses that can be detected through static analysis of the program code, while taking into consideration the DL fundamentals, best practices, and DL library's intricacies.

- The majority of silent DL bugs come from computations errors or misconfigurations that can be detected by monitoring the training program's behavior in order to detect unstable learning dynamics or inefficient trained models.

- The generation of synthetic inputs that are valid and optimized for test adequacy criteria, is limited by the large and complex input space of DL applications. In contrast, crafting the transformations needed to derive these novel inputs using a sample of the original data simplifies the validation process and mitigates the curse of high dimensionality.

- Using semantically-preserving transformations, test inputs are derived to assess the robustness of the model. Domain knowledge can be utilized to build input transformations that assess the consistency of model behaviors to foreknown properties and characteristics.

- If distributional shifts are significant, the use of statistically-learned features or uncertainties for out-of-distribution(OOD) detection can be misleading. Apriori system properties, derived deductively from domain knowledge, may provide more stable and robust OOD input recognition.

Leveraging concepts from software testing, deep learning, and domain knowledge, this thesis contributes to three important aspects of DL software quality assurance: (i) automated debugging methods specialized for DNN training programs, including both static code inspection and dynamic behavior monitoring; (ii) search-based and domain-aware DL testing approaches that assess application-specific robustness, consistency with physics-grounded sensitivities, and elusive quantization inefficiencies; and (iii) smoothness-based out-of-distribution (OOD) detection strategy that identifies suspicious DNN behaviors against shifted and likely OOD inputs based on a common apriori system property like smoothness.

## 1.5 Thesis Overview

1. *A static rule-based DL program debugger.* We develop a static debugger that abstracts the learning program skeleton and components into a meta-model. Then, by running a technologically-aware parser module on the DL code, our static debugger instantiates the meta-model and creates a compiled object model for the underlying learning program that allows it to detect faults and design inefficiencies as violations of designed model-based verification rules. In the evaluation, we implement a tool covering mainstream DL toolkits such as Tensorflow and Keras in order to assess the effectiveness of our approach on real-world buggy programs.

2. *A dynamic property-based DL program debugger.* We build a dynamic debugging approach that aims to bridge theoretical and practical knowledge in order to equip expert and especially non-expert DL practitioners with automated verification routines that validate, starting from the model construction and data loading, the fundamental DL principles; then, watch continuously the training dynamics during the execution to make sure of the stability and proper convergence of the data fitting process. For assessment, we implement a Tensorflow-based tool to test our elaborated checks on real-world learning programs.

3. *A search-based DL testing approach.* We construct a generic-purpose DL testing approach to fill in the gap between iid evaluations and required application-specific generalization. By searching over the transformations instead of the inputs, we develop practical metamorphic testing pipelines for many DL domains. Then, we use and assess various nature-inspired, population-based metaheuristics that are suitable for driving the generations of data transformations towards diverse regions with high fault-detecting capabilities. DeepEvolution is examined for its potential to expose hidden weaknesses of both optimized and compressed DNNs to maintain, respectively, reliable and stable predictions when applied to regular and unusual usage scenarios.

4. *A physics-based adversarial machine learning.* We propose a physics-guided model testing that formulate the physics first principles and system-related design properties in the format of input-output sensitivities to expose physics inconsistencies of the model's mapping function. Next, a search-based approach was applied to explore constrained neighbors of each genuine input aiming at finding those on which the model violates the foreknown physics-grounded sensitivities. Moreover, all the revealed counter-examples should be exploited by physics-informed regularization techniques that perform fine-tuning to constrain the neural network's optimization with the desired level of physics

consistency. Our proposed physics-based adversarial ML is evaluated on two DL-based performance models for aircraft systems, for which the sensitivity rules can be derived beforehand by aircraft engineers.

5. *A smoothness-based out-of-distribution detector.* We introduce our smoothness-based OOD detection strategy that ensures selective and reliable predictions on only the ID inputs. Specifically, we create pointwise sensitivity profiles that can distinguish ID and OOD samples based on their adherence to smooth domain regions. Our method is deployed as a novel criterion for the systematic switch between the surrogate and its HF model counterpart in hybrid optimization settings. The evaluation was conducted on three aircraft design variables study cases with regard to the effects on design assessment errors and overhead costs of HF model requests.

### 1.6 Thesis contributions

- Our static DL program debugger enables detecting the bugs through code inspection inexpensively, and without even a single training iteration. It has covered 23 common errors and poor design practices of DL training programs, and provides associated detection rules that successfully finds 64 faults and design inefficiencies in 34 real-world DL programs extracted from Stack Overflow posts and GitHub repositories. Hence, our approach reaches a recall of 70.5% and a precision of 100% in revealing both DL faults and design issues.

- Our dynamic DL program debugger defines a catalog of 21 pitfalls relative to training program's components including data loader, parameters, activations, optimization, and regularization. We demonstrate how these training pitfalls can be captured on-running by our designed verification routines, operating on intermediate DNN states and optimization dynamics. Its effectiveness was evaluated in exposing both DL coding bugs and misconfigurations with (precision, recall), respectively, equal to (90%, 96.4%) and (77%, 83.3%). Its comparison with Amazon Sagemaker Rule-based Debugger(*SMD*) shows that it outperforms *SMD* by detecting 75% rather than 60% of the total of reported bugs in real-world buggy programs extracted from StackOverflow and GitHub. Its usability evaluation was conducted in collaboration with two DL engineers, and they were able to successfully locate and fix 93.33% of bugs contained in 10 buggy training programs.

- Our search-based DL testing approach succeeds in revealing DNN erroneous behaviors. Precisely, it achieves, averagely, 41%, 24.5%, and 5% of misclassification detection

rates for the studied DL domains, respectively, image, audio, and text. DeepEvolution was also able to expose hidden quantization inefficiencies. Specifically, it reaches, on average, 21.5%, 24%, and 2% of divergence exposure rates when comparing genuine and quantized DNNs designed for visual, speech, and natural language text recognition. Throughout all the study cases, DeepEvolution outperformed its competing alternative, TensorFuzz, which is Google Brain's coverage-guided fuzzing framework specialized for DNNs.

- Our physics-guided adversarial testing method was able to detect DNN violations of three types of physics-grounded sensitivity rules through the following assertions: (i) invariance test (i.e., output remains constant under input perturbation); (ii) directional expectation tests (i.e., decreasing or increasing output under input perturbation). Next, our physics-informed adversarial training technique allows fine-tuning the parameters of the DNN to update the inductive biases towards complying with the physics grounded sensitivities, while maintaining a low error rate in predictions. Based on two industrial case studies of aircraft system performance models, we show that our adversarial testing using Genetic Algorithm(GA) was capable of generating up to hundreds of physics inconsistencies. Then, its follow-up physics-guided regularization succeeded in fixing on average 88.5% of the detected violations, which leads to an average of 6% gain in the predictive performance on the original test data.

- Our smoothness-based OOD detector demonstrates the potential of exploiting apriori system properties like smoothness to effectively expose OODs, thereby avoiding the pure data-driven predictions and uncertainties that are often misleading and overconfident. We carry out an industrial assessment on three surrogate aircraft design models, concluding that SmOOD does cover averagely 85% of actual OODs on all the study cases, and when SmOOD is set up as a surrogate/HF switcher in hybrid surrogate optimization settings, we obtain a decrease error rate of 34.65% and a computational speed up rate of 58.36×, averagely.

## 1.7 Organisation of the thesis

The rest of this thesis is organised as follows:

- Chapter 2 presents the fundamental concepts and methods related to deep learning trustworthiness that are needed to understand our research works.

- Chapter 3 outlines the literature review of the related studies and research works.

- Chapter 4 describes our static rule-based debugging method for DNN training programs: design, implementation, and assessment on DNN programs using TensorFlow and Keras.
- Chapter 5 presents our dynamic property-based debugging approach for DNN training programs: design, implementation, and evaluation on real-world TensorFlow-based DNN programs.
- Chapter 6 details our search-based DL testing approach for metamorphic transformation generation, its Tensorflow-based implementation, and its evaluation results on visual, speech and natural language recognition models.
- Chapter 7 outlines our physics-based adversarial ML: testing and regularization phases, as well as their effectiveness on aircraft system performance models.
- Chapter 8 presents our smoothness-based OOD detection method for DNNs and its performance evaluation on hybrid surrogate aircraft design optimization.
- Chapter 9 summarizes and concludes the thesis, and discusses the limitations and future work.

# CHAPTER 2 BACKGROUND

**Chapter Overview** Section 2.1 introduces the concepts and methods of Deep learning. Section 2.2 describes the concepts and methods of software quality assurance, especially for programs without oracle. Finally, Section 2.3 concludes the chapter.

## 2.1 Deep Learning

Over the last decade, the growing amount of data and the advent of GPUs have enabled the application of deep learning in automation tasks across various domains. In this section, we describe the deep neural networks (DNNs), which are the backbone behind DL, their engineering process and challenges, as well as their software development.

### 2.1.1 Deep Neural Networks

A deep neural network (DNN) is an artificial neural network with a stack of multiple computational layers, hence the adjective "deep". DNNs are often much harder to train than shallow neural networks [24]. However, they are endued with a hierarchical features learning that lets them capture increasingly complex patterns directly from the data when they are appropriately designed and trained. DNNs include many variants of architectures that have found success in several domains. We present in the following the Deep Feed-Forward Neural Network and its popular variant Convolutional Neural Network.

#### Deep Feed-Forward Neural Networks

Feedforward neural network (FNN) architecture is the quintessential and the most used neural network [25]. The objective of FNNs is to learn the mapping of a fixed-size input (for example, a signal vector) to a fixed-size output (for example, a probability for each label). Apart from the input and output layers, FNN contains a cascade of multiple intermediate layers, which are called hidden layers because the ground truth data does not provide the desired values for these layers. Last, the name feedforward arises from the fact that information flows through the processing layers in a feed-forward manner, i.e., from input layer, through hidden layers and finally to the output layer. Figure 2.1 illustrates a simple FNN.

A FNN encapsulates a mapping function $f$ that maps the input $x$ to its corresponding output $y$ as presented in Equation 2.1. If $y$ is a category label, the FNN solves a classification problem

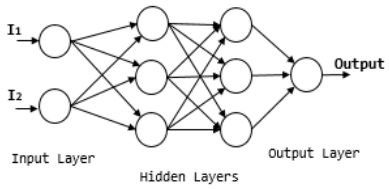



Figure 2.1 Schema of feedforward neural network

and if $y$ is a continuous value, the FNN performs a regression task.

$$y = f(x; W, b) \tag{2.1}$$

Where $W$: weights and $b$: biases represent the learnable parameters.
Given the training data $D = \{(x_i, t_i); \forall i \in [1, n]\}$, the FNN training algorithm aims to find the best approximation function $f^*$ by learning the optimal values of its inner parameters $W^*$ and $b^*$. To do that, a loss function, $loss(f, x, t)$ is leveraged to measure how well the prediction $y = f(x)$ matches the ground truth output $t$. Considering the example of multinomial classification problem, we have $y$, a vector of probabilities where $y(j)$ represents the probability that $x$ belongs to the label $j$ and $t$, a one-hot encoding label where $t(j) = 1$ if $j$ is the true label; otherwise $t(j) = 0$. The most common loss function is the cross entropy $loss(f, x, t) = -\sum_j t(j) \log(y(j))$.
To estimate the loss over all the training data $loss(f, D)$, we use an expectation, as formulated in Equation 2.2, that can be either the average or the sum of data instances' losses.

$$loss(f, D) = E_i[loss(f, x_i, y_i)] = E_D[loss(f, D)] \tag{2.2}$$

Therefore, the optimal parameters $W^*$ and $b^*$ result in the best function approximation that spawns the minimum possible estimated loss in the training data $D$.

$$W^*, b^* = \underset{W,b}{\operatorname{argmin}} E_D[loss(f^*, D)] \tag{2.3}$$

The loss minimization problem can be solved, iteratively, using gradient descent algorithm, where the following equations represent an iteration's updates :

$$W^{(t+1)} = W^{(t)} - \eta E_D[loss(f^{(t)}, D)]W; \quad b^{(t+1)} = b^{(t)} - \eta E_D[loss(f^{(t)}, D)]b \tag{2.4}$$

As introduced, the FNN assembles multiple computational layers and each layer $l$ perform a computation; so it encapsulates a kind of sub-function $f_l(x; W_l, b_l)$ with inner parameters $W_l$

and $b_l$. Thus, the approximate mapping function $f$ of an FNN with $L$ layers is a composite function formulated as below :

$$f(x; W, b) = f_L(f_{L-1}(...(f_1(x; W_1, b_1)...); W_{L-1}, b_{L-1}); W_L, b_L) \tag{2.5}$$

Where $W = \{W_l, \forall l \in [1, L]\}$ and $b = \{b_l, \forall l \in [1, L]\}$.
Given the huge number of parameters to approximate, the computation of gradients would be very time-consuming. Deep learning relies on a fast algorithm, named backpropagation, that applies the derivative chain rule principle to compute, sequentially, all these derivative backing from the output to the first hidden layer, while taking full advantage of the derivatives estimated w.r.t previous layers. Backpropagation is based on two alternatives main phases, respectively, forward pass and backward pass, which are detailed below.
**Forward Pass.** Each hidden layer $l$ contains a set of computation units, called neurons, that perform a linear transformation $z_l$ of their inputs from previous layers and pass the result through an activation function $\Phi_l$. The latter is a non-linear transformation $a_l$ that allows adding non-linearity in the approximated mapping function in order to be insensitive to irrelevant variations of the input. The layer's computation can be written as:

$$z_l = W_l a_{l-1} + b_l, \forall l \in [1, L] \tag{2.6}$$

$$a_l = \Phi_l(z_l), \forall l \in [1, L] \tag{2.7}$$

We note that $a_0$ which is the input layer activation is equal to the input data $x$. The hidden layers share generally the same activation function. We denote them $\Phi_1 = \Phi_2 = ... = \Phi_L = \Phi$. We denote the activation function of the output layer by $\Psi = \Phi_L$.
**Backward Pass.** First, we introduce an intermediate quantity, $\delta$, where $\delta_l$ is the vector of error associated to the layer $l$. $\delta_l$ is computed as the gradient of the loss with respect to the weighted input $z_l$. In the following, we present the equations used by the backpropagation algorithm to compute the error for every layer. We refer the reader to the work of Goodfellow et al. [25] for the proof and in-depth details.

$$\delta_L = lossz_L = \nabla_a(loss) \odot \Psi'(z_L); \quad \delta_l = lossz_l = W^{(l+1)T}\delta_{l+1} \odot \Phi'(z_l) \tag{2.8}$$

In Equation 2.8, $\odot$ is the Hadamard product, which is an elementwise product of two vectors in a way that for each component $j$, $v \odot u$ means $(u \odot t)_j = s_j t_j$.
Second, we formulate the gradient of loss w.r.t the DNN parameters using the computed error term $\delta$.

$$lossW_l = \delta_l a^{(l-1)T}; \quad lossb_l = \delta_l \tag{2.9}$$

Last, we perform both of weights and biases iteration updates in the opposite direction of their gradients w.r.t the loss.

$$W^{(i+1)} = W^{(i)} - \eta\delta_l a^{(l-1)T}; \quad b^{(i+1)} = b^{(i)} - \eta\delta_l \tag{2.10}$$

In practice, the backpropagation algorithm does not loop over the training examples and perform the forward and backward passes on each example separately. Indeed, it relies on mini-batch stochastic gradient descent that computes both passes on a mini-batch of examples simultaneously. This is done by formulating the equations presented above as fully matrix-based formulas given the input matrix $X = [x_1, x_2, ..., x_m]$ of a mini-batch containing $m$ examples. Thus, the parameters updates are based on the average of loss gradients estimated over all the examples of the batch, which leads to more stable gradient updates.

### Deep Convolutional Neural Networks

Convolutional Neural Network (CNN) represents a particular type of feedforward network that is designed to process data in the form of multiple arrays, such as 2D images and audio spectrograms, or 3D videos [25]. A CNN contains the following specialized layers that transform the 3D input volume to a 3D output volume of neuron activations: Convolutional Layer, Activation Layer, and Pooling Layer.

**Convolutional Layer.** The main building block of this type of transformation layer is the convolution. A convolution provides a 2D feature map, where each unit is connected to local regions in the input data or previous layer's feature map through a multi-dimensional parameter called a filter or kernel. These filters play the role of feature detectors. The feature map is produced by sliding the filter over the input data, then computing products between the filter entries and the local input region at each spatial position, to infer the corresponding feature map response. Different filters are performed in a layer and resulting feature maps are stacked in 3D volumes of output neurons. The separate filters aim to detect distinctive local motifs in the same local regions. However, all units in one feature map share the same filter, because motifs are generally invariant to location.

**Activation Layer.** As in any FNN layer, we use an activation function to add non-linearity to the computed value. The activation layer applies the activation function on the extracted feature map as an element wise operation (i.e., per feature map output). The resulting activation map indicates the state of each neuron, i.e., active or not.

For each hidden neuron $(i, j)$ in a feature map:

$$a_{ij} = \Phi(z_{ij}); \tag{2.11}$$

**Pooling Layer.** The pooling layer ensures spatial pooling operation to reduce the dimensionality of each feature map and to retain the most relevant information by creating an invariance to small shifts and distortions. Depending on the chosen spatial pooling operations, it can be average or max pooling that computes, respectively, the average or max of all elements in a pre-defined neighboring spatial window size. Therefore, these neighboring pooling makes the resulting activation maps smaller and robust to irrelevant variance, which helps shortening the training time and controlling the overfitting.
Figure 2.2 shows a typical architecture of CNN with two main stages: (1) multiple stack of convolution, activation, and pooling that ensure the detection of relevant features from the input data; (2) the final activation map is flatten to be a vector of features and is fed to a fully-connected neural network that performs the prediction on top of these extracted features to estimate the labels' scores or the predict value.

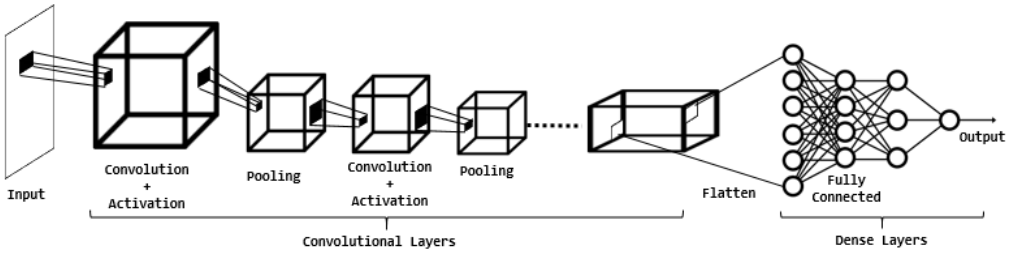



Figure 2.2 Schema of convolutional neural network

### Other Mainstream DNN Architectures

Recurrent neural networks (RNN) [25] learn patterns from sequential inputs like sequence of words or time series signals. RNN uses the same parameters at every step, in addition to the features found at earlier ones, in order to search local features on the current features. As a result of repeated multiplication with the same weight, they suffer from vanishing gradients. The most common way to circumvent the vanishing problem is using long-short-term memory (LSTM) and its derivatives. Indeed, the gating mechanisms built into LSTMs (forget, update, and output gates) control information flows across steps and keep track of the features relevant to sequence assessment. Moreover, a new class of DNN was recently introduced called Transformers [26]. The latter represent encoder-decoder networks that incorporate attention mechanisms to focus on the relevant parts of high-dimensional inputs. First, they encode the input vectors into positional embeddings. Then, they compute the pairwise inner product between each pair of embeddings to search systematically for a set of positions in the encoder provided states where the most relevant information is available.

Therefore, the decoder phase leverages the extracted features to predict the target output.

### 2.1.2 DNN-based Model Engineering

First, DL engineers collect labeled data that incorporate the knowledge required to perform the target task. The collected data is divided into three different datasets: *training dataset*, *validation dataset*, and *testing dataset*. After readying these data sets, DL engineer designs and configures the DNN by choosing the architecture, setting up initial hyperparameters values, and selecting variants of mathematical components that include activation functions, loss functions, and gradient-based optimizers. DL engineer should consider application requirements, data complexity, and best practices or guidelines from other similar works. After preparing the DL system ingredients (*i.e.,* data and model design), the training process starts and systematically evolves the decision logic learning towards effectively resolving the target task. Indeed, training a DNN-based model using an optimizer consists in gradually minimizing the loss measure with respect to the training dataset. Once the model's parameters are estimated, hyperparameters are tuned by evaluating the model performance on the *validation dataset*, and selecting the next hyperparameters values according to a search-based approach that aims to optimize the performance of the model. This process is repeated using the newly selected hyperparameters until a best-fitted model is obtained. Last, this best-fitted model is tested using the testing dataset to verify if it meets the predictive performance requirement. Throughout the above-explained model engineering process, we show how DNN automatically derives the computational solution to a specific task from the data; however, these complex models that handle high-dimensional data with no prior feature engineering can be difficult to train. Hence, we describe in the following the DNN training challenges and simple and advanced regularization techniques proposed for DNN.

### 2.1.3 Difficulty of training DNNs

Traditionally, machine learning algorithms design the loss function and constraints carefully; ensuring that the minimization problem is convex, where any found local minimum is guaranteed to be a global minimum. When training a deep neural network, the loss function is not only non-convex but also tends to have a large number of "kinks", flat regions, and sharp minima [27]. This makes the loss minimization problem poorly-conditioned, and consequently, makes it hard to solve. Indeed, conditioning refers to how swiftly a function changes with respect to small variations in its entries [25], hence, high sensitive functions lead to poor conditioning. In fact, the Hessian matrix of the loss encapsulates all the second-derivatives that represent the curvature of the loss surface. The condition number which is the ratio

between the largest and the smallest eigenvalue is very important because a high condition number indicates a situation of ill-conditioning, where some parameters have huge curvature while others have smaller one. Such a situation results in a pathological loss curvature, and a first-order gradient descent would have difficulty progressing. Nevertheless, practitioners succeed to train modern DNNs using first-order gradient-based optimization. This practical trainability success is highly dependent on the design of DNN, the choice of optimizer, parameters initialization, normalization, regularization, and a variety of hyperparameters. However, finding adequate configurations and parameters can be very challenging and the optimization of neural networks is still an open problem. In practice, training a DNN for a novel problem, context or data requires a series of trial-and-error using different configuration choices and hyperparameters tuning. These configuration choices have a strong effect on the conditioning of the minimization problem. However, novel DNN design components like skip-connections of ResNet, advanced regularization and reparameterization techniques [28, 29] have shown an ability to improve the Lipschitzness of the loss function, which means a loss exhibiting a significantly better smoothness. These smoothing effects impact the performance of the training algorithm in a major way because it provides more confidence that the estimated gradient direction for each training step remains a fairly accurate estimate of the actual gradient direction after taking that step. This enables performing update steps without high risk of running into a sudden change of the loss landscape; including flat region (corresponding to vanishing gradient) or sharp local minimum (causing exploding gradients). In other words, finding good ways to configure and parametrize the DNN ensures the stability of the loss function and better predictiveness of its computed gradients. Below, we explain in detail the regularization techniques used for DNN.

### 2.1.4 Regularization of DNN Training

Given the high capacity of DNNs, developing regularization techniques that stabilize the training evolution and prevent the model from overfitting the training data, has been one of the major research efforts in the deep learning field.

#### Standard Regularization Techniques

The standard regularization techniques consist in adding restrictions on the values of the trained parameters, i.e., by adding a penalty term $\Omega(W)$ in the loss function $loss(f^*, D)$ (see the regularized loss from Equation 2.12) that can be seen as a soft constraint on the

magnitude of parameters to restrict and smooth their corresponding distributions.

$$\tilde{loss}(W, b, D) = loss(W, b, D) + \lambda\Omega(W) \tag{2.12}$$

Where $\lambda$ fixes the relative contribution of the norm penalty omega; so setting $\lambda = 0$ means no regularization and larger values of $\lambda$ correspond to more regularization. The most popular penalty consists in penalizing the weights of the linear computations; keeping their values closer to the origin by using L2-norm ($\Omega(W) = \frac{1}{2}W_2^2$) and–or enhancing the sparsity of the weights by using L1-norm ($\Omega(W) = W_1$). With the gradient-based learners, we compute the gradient of the weight penalty term $\Omega(w_i)$ w.r.t weight $w_i$ as described in the following formula.

$$\frac{\partial\Omega}{\partial\mathbf{w}_i} = \begin{cases} 2 \times \mathbf{w}_i^{(t)}, & \text{if } \Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2 \\ 1 \times \text{sign}\left(\mathbf{w}_i^{(t)}\right), & \text{if } \Omega(\mathbf{w}) = \|\mathbf{w}\|_1 \end{cases} \tag{2.13}$$

From the above formula, we can see that L2-norm penalty continuously reduces the magnitude of the weights proportionally to it while L1-norm reduces the magnitude by a constant. Thus, L2-norm and L1-norm pushes, respectively, the magnitude of weights towards increasingly lower and zero values. Moreover, the defined hyperparameter, $\lambda$, controls the strength of the applied regularization. It is therefore important to adjust it appropriately, taking into account the design of the model and the complexity of the target problem but concerning changes w.r.t size of batches, it has been shown that scaling the $\lambda$ by $1/m$, where $m$ is the size of the batch, can make it comparable across different size of batches [30], which avoids tuning manually $\lambda$. Intuitively, the training algorithm tries to approximate an unknown distribution by minimizing the empirical error on a sample of data; so a large sample is likely to be more representative of this unknown distribution, and as a result, less regularization might be needed in order to capture the maximum information about the target data distribution.

### Advanced Regularization Techniques

The advanced regularization techniques for DNNs include normalization and stochasticity to the DNN inner computations [31]. In fact, manipulating randomly the DNN architecture, over training passes, minimizes the risks that the learned parameters are highly customized to the underlying training data. Furthermore, normalizing the input of each layer, not only the input data, improve furthermore the smoothness of the loss landscape towards more stable gradient-based optimization, and fortunately, higher chances to avoid saddle points and ineffective local minima.

**Dropout.** One of the most frequently used regularization techniques is dropout [32], which randomly deactivates a subset of neurons from the dropped dense or convolutional layer at

every training iteration. Indeed, the degree of randomness of dropout should be adjusted with respect to the width of the layer using a hyperparameter, *pkeep*, which represents the neurons' retention probability. At inference time, dropping neurons is stopped and compensated by multiplying all the weights in the layer by *pkeep* in order to keep the distribution of the layer outputs (i.e., results of the layer's affine transformation) during inference time close to the distribution during training time. Mathematically, for each hidden layer $l$, we define a binary mask $m^l$, where each element can be 1 with predefined probability *pkeep*. Thus, the dropped out version of the hidden layer output $z^l$ masks out units using element-wise production, $z_d^l = m^l \odot z^l$. Intuitively, dropout reduces the risk of overfitting by making the DNN robust against the deactivation of some neurons, which forces the DNN to rely on population behavior instead of the activity of other feature detectors unit (i.e., preventing the co-adaptation of feature detectors). Indeed, model ensembling, a well-known technique in statistical learning, consists in combining the output of multiple models, each trained differently in some respect, to generate one final answer. The resulting improvements on the performance metrics explain its domination over recent machine learning competitions [25], however, it requires a much larger training time by definition (compared to training only one model). More fundamentally, dropout [33] simulates model ensembling without creating multiple neural networks by combining the predictions of multiple sub-DNNs resulting from dropping, randomly, different subset of neurons between every two consecutive training passes.

**Batch-normalisation.** Another interesting regularization strategy designed for DNN is Batch-normalisation (Batchnorm). It relies on continuously normalizing intermediate activations across batches during a mini-batch loss minimization [34]. Indeed, the proceeded normalization is based on the standardization of each intermediate feature using the pre-computed mean and standard deviation on the current batch, respectively, $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$ and $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ given the batch data $B$ of size $m$. The normalized activations $\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ would have zero mean and unit standard deviation. However, the resulting reduction of magnitude may induce information loss caused by the distortions in the learned intermediary features, and consequently, the degradation of model's learning capacity. Therefore, batch-normalization performs a linear transformation to scale and shift the normalized activations, $a_i = \alpha\hat{x_i} + \beta$ with aim of preserving the expressiveness of the DNN through learning additionally both of the parameters $\alpha$ and $\beta$. In addition, batch-normalisation also computes two other statistics, $E[x]$ and $Var[x]$, which represent, respectively, the moving average and the moving variance of the flowing data during all the training. Thus, at the test time, we use population mean($E[x]$) and population variance($Var[x]$) to standardize the layer outputs instead of using batch mean($\mu_B$) and batch variance($\sigma_B^2$).

Santurkar et al. [5] investigated the fundamental reasons behind the effectiveness of batch normalization in regularizing modern DNNs. They found that normalizing continuously all the inputs of hidden layers (i.e., all the activations) instead of normalizing only the inputs, makes the surface of loss smoother, which ensures faster convergence and safer training using relatively higher learning rates for which unnormalized DNNs (i.e., standard variants without batchnorm) diverge. This positive effect on the loss landscape is called better conditioning of loss minimization problem, which is described explicitly in the following sub-section.

### 2.1.5 DNN-based Software Development

DNN-based programs are implemented as traditional software using programming languages such as Python, but include high-dimensional algebraic computations and automatic differentiation of derivatives. Due to the growing amount of training data and the increasing depth of DNN architecture, their implementations should support parallel and distributed execution in order to leverage the power of high-performance computing hardware architectures. Recent advances in Deep Learning have led to the release of several DL software libraries such as Theano [35], Torch [36], Caffe [37], and Tensorflow [38] in order to assist ML practitioners in developing and deploying state-of-the-art DNNs. These libraries provide optimized implementations of compute-intensive and parallel algebraic calculations that take full advantage of high-performance and distributed hardware infrastructures. Most DL engineers today leverage existing DL frameworks to encode the designed DNN into programs. Ready-to-use features and routines offered by DL libraries often have to trade off between the coverage of novel DL functionalities and the ease of rapid implementation and extension of DNN software prototypes. As a compromise solution, they uniformly include, for each newly-implemented DL functionality, a bundle of automated steps and default settings following its common usage trends in the community. This enables quick prototyping of regular DNNs while keeping the flexibility to try other configurations with the tweakable setting options available for every provided DL routine. As a consequence, DL developers should be aware of the intricacies of these DL libraries to choose the appropriate configurations and avoid breaking their implicit assumptions in regard to the usage of their built-in routines.

## 2.2 Software Quality Assurance

Software quality assurance (SQA) consists of standards, procedures, and activities that can be employed during various stages of the software development process to ensure the quality of the delivered software product. Basically, SQA incorporates and implements software testing and program verification methodologies to identify errors, bugs, or missing requirements. In

this way, SQA activities aid the development process in reducing bugs beforehand, preventing hidden defects from occurring during deployment. Below, we detail the main methods of SQA pertinent to the present thesis.

### 2.2.1 Software Testing

Software testing requires the generation of test inputs. There have been efforts to automate the process of generating test inputs in order to circumvent the combinatorial explosion of all possible inputs for large size programs, which has proven that its exhaustive enumeration is an undecidable problem [39]. In the following, we describe two established generation techniques, which are recently used for DL testing.

#### Pseudo Test Oracle

When it is well-defined based on methods introduced below, a pseudo test oracle [40] is able to distinguish a program's correct behavior from an incorrect behavior. Moreover, those automated pseudo oracle can be improved, over time, to become more effective in detecting inconsistencies.
We detail some standard pseudo-oracle that have been adapted for testing DL systems:

**Metamorphic Testing** Metamorphic testing [41] is a derived test oracle that allows finding erroneous behaviors by detecting violations of identified metamorphic relations (MR). The first step is to construct MRs that relate inputs in a way that the relationship between their corresponding outputs becomes known in prior, so the desired outputs for generated test cases can be expected. For example, a metamorphic relation for testing the implementation of the function $\sin(x)$ can be the transformation of input $x$ to $\pi - x$ that allows examining the results by checking if $\sin(x)$ differs from $\sin(\pi - x)$. The second step is to leverage those defined MRs in order to automatically generate partial oracles for follow-up test cases, *i.e.,* genuine test inputs could be transformed with respect to one MR, allowing the prediction of the desired output. Therefore, any significant differences between the desired and the obtained output would break the relation, indicating the existence of inconsistencies in the program execution. As the example of $\sin(x)$, breaking the relation between $x$ and $\pi - x$ stating that $\sin(x) = \sin(\pi - x)$ indicates the presence of an implementation bug without needing to examine the specific values computed through execution. The efficiency of the metamorphic testing depends on the used MRs. MRs can be identified manually from specific properties of the algorithm implemented by the program under test, or inferred automatically by tracking potential relationships among the output of test cases run that hold across

multiple executions. However, automatically generated MRs need to be analyzed by an expert to ensure that they are correct considering the domain knowledge.

**Differential Testing** The differential testing [42] creates a partial test oracle that detects erroneous behaviors by observing whether similar programs yield different outputs regarding identical inputs. The intuition behind this approach is that any divergence between programs' behaviors, solving the same problem, on the same input data indicates the presence of a bug in one of them. It is quite related to $N$-version programming that aims to produce, independently, alternative program versions of one specification, so that if these multiple program versions return different outputs on one identical input, then a "voting" mechanism is leveraged to identify the implementations containing a bug. Davis and Weyuker [40] discussed the application of differential testing for 'non-testable' programs. The testing process consists of building at first multiple independent programs that fulfill the same specification, but which are implemented in different ways, *e.g.,* different developers' teams, or different programming languages. Then, we provide the same test inputs to those similar applications, which are considered as cross-referencing oracles, and watch differences in their execution to mark them as potential bugs.

**Property-Based Testing** Property-based testing (PBT) is a practical testing method [43] that provides a systematic way for reasoning about the properties of the appropriate program's behaviors instead of the correct outcomes. For example, one may validate that a random data generator can produce probabilities within $[0, 1]$, while abstracting away from the actual probabilities. It follows the philosophy of invariant detection, which defines a set of invariant properties that allow aligning an incorrect execution against the expected execution [44]. Indeed, PBT defines the essential properties that the under-test program must respect in any possible execution scenario, rather than searching for the exhaustive set of all valid input-output pairs. These properties should represent high-level specifications, describing the program's correctness. First, PBT requires the collection of sufficient properties about the component under test (such as its function, program, or system). Then, the verification process starts by generating inputs for the component relying on specific heuristics to cover the equivalent classes of data inputs. Afterward, it validates for all the generated inputs that all preconditions, invariant properties on intermediary results, and postconditions associated with the component under test are totally true. When a property is failed, the counterexample is shrunk by searching the minimal combination of input elements that causes the property to fail. In fact, large inputs may cause the failure of multiple properties; so shrinking the input allows developers to extract the smallest part of inputs that is capable

of reproducing a particular failure, which is essential for fixing the bug.

### Test Adequacy Assessment

Test adequacy assessment consists of estimating to which extent the generated test cases are 'adequate' to terminate the testing process with confidence that the program under test is implemented properly. In the following, we introduce widely-used techniques for the assessment of test adequacy and enhancement of test sets, from which researchers borrowed inspiration for testing DL systems.

**Code coverage :** it includes different test adequacy criteria that measure the proportion of the program's source code that is executed by test cases. It helps assessing the amount of internal logic triggered when testing the program. For example, one can use statement coverage (*i.e.,* run each statement at least one time), branch coverage (*i.e.,* try each branch from any decision point at least once), and path coverage (*i.e.,* test the different possible sequence of decisions from entry to the program exit). Indeed, test cases achieving high coverage are more likely to uncover the hidden bugs because the portions of code that have never been executed have a higher probability of containing defects and to work improperly.

**Mutation Testing :** it assesses the effectiveness of the test cases in revealing faults intentionally introduced into the source code of mutants [45]. Indeed, test cases should be robust enough to kill the mutant code (*i.e.,* the mutant failed the test). A mutation is a single syntactic change that is made to the program code; so each mutant differs from the original version by one mutation. The latter can be categorized into 3 types: (1) statement mutation: changes done to the statements by deleting or duplicating a line of code; (2) decision mutation: one can change arithmetic, relational, and logical operators; (3) value mutation: values of primary variables are modified such as changing a constant value to much larger or smaller values. The ratio of killed mutants against all the created mutants is called the mutation score, which measures how much the generated test cases are efficient in detecting injected faults.

Test adequacy assessment is required to guide the test generation toward optimizing the testing effort and producing effective test cases that enhance the chances of detecting defects. The test data generation process is the subject of the next subsection.

### Test Data Generation

Generating test inputs for software is a crucial task in software testing. In the earliest days, test data generation was a manual process mainly driven by the tester, which inspects the specification and the code of the program. However, this traditional practice is costly and laborious since it requires a lot of human time and effort. Researchers have worked to automate the test inputs generation process, but automation in this area is still limited. In fact, exhaustive enumeration of all possible inputs is infeasible for large size programs and previous attempts to automate the generation of inputs for real-world software applications have proven that it is an undecidable problem [39].

**Coverage-guided Fuzzing (CGF) [46]:** A fuzzing process maintains an input corpus containing inputs to the program under test. Mutations are applied randomly to those inputs, and coverage-guided fuzzing retains only those mutated inputs that trigger new structural coverage, such as statement or branch coverage, in the corpus. CGF has proven highly effective at detecting errors in traditional software. Indeed, AFL and libFuzzer are two widely-used fuzzers that succeed to expose serious bugs in real software systems.

**Search-based Software Testing (SBST) [47]:** It formulates the code coverage criteria as a fitness function, which can compare and contrast candidate solutions from the space of possible inputs in terms of the covered portions of the code. Using this fitness function, SBST leverages metaheuristic search techniques, such as Genetic Algorithm, to drive the search into potentially promising areas of the input space; generating effective test cases and increasing the code coverage.

**SBST for DL Testing:** Since we propose a search-based approach for DNN models, we detail furthermore this software testing approach. The latter uses metaheuristics [48] that represent computational approaches, solving an optimization problem by iteratively attempting to ameliorate a candidate solution with respect to a fitness function. They require only few or no assumptions on the properties of both the objective function and the input search space. However, they do not provide any guarantee of finding an optimal solution. Their applicability in structural testing is suitable as these problems frequently encounter competing constraints and require near optimal solutions, and these metaheuristics seek solutions for combinatorial problems at a reasonable computational cost. Specifically, nature-inspired population-based metaheuristics possess intrinsically complex routines and non-determinism that make them a high potential candidate for spotting vulnerable regions in the large, multi-dimensional input space of the DL models. In the following, we describe two widely-used metaheuristic algorithms in testing deep learning software systems. Indeed, they succeed in crafting black-box adversarial examples efficiently with few queries and they have achieved

white-box comparable performances in different application domains including computer-vision [49–52], natural language processing [53–55], and speech assistance DNNs [56].

**Particle Swarm Optimization(PSO) [57]** mimics the behavior of a swarm of birds to search a very large space of candidate solutions. PSO maintains a population of candidate solutions called particles. As particles move in the search space, they seek better solutions to the problem based on their fitness values. The inertia weight, w, affects particle velocity and search space expansion. The movement of each particle within an iteration is guided by its local best position, $p_{best}$, which refers to exploring its best neighbour regions, while at the same time being guided toward a global best position¸ $g_{best}$, by all particles, which refers to exploiting the highest fitness region found. The PSO algorithm employs two trust coefficients, $\varphi_p$ and $\varphi_g$, which set up a particle's confidence in itself (cognitive coefficient) and in its neighbors (social coefficient).

**Genetic Algorithm(GA) [58]** emulates the behavior of biological evolution, including basic selection, crossover, and mutation operations that can lead to, potentially, better individuals in every generation. By using a tournament selection strategy, GA selects a $r_{parents}$ of individuals to become parents for the next generation. Breeding is done by randomly picking matching pairs of parents and using one of these binary crossovers [59], one-point, two-point, or uniform, to produce new offspring. Each parent in a couple will be assigned a relative importance based on its fitness level. At the end of the breeding, we apply mixed random mutations to alter the features of the offspring in order to maintain and introduce diversity into the new generation. Indeed, every descendant can be subject to a mutation, depending on a fixed probability, $p_{mutation}$.

### 2.2.2 Software Verification

Software verification can be done automatically through model checking that verifies whether a system's finite-state model satisfies a given specification [60–63]. Indeed, model checking investigates all the reachable states of a model that are expressed formally and checks whether some given property is satisfied. If the given property is satisfied, a witness will be generated, i.e., a path starting from the initial state and leading to the state in which the property is satisfied. Otherwise, in the case that the property is not satisfied, which is called refutation, a counterexample will be generated, i.e., a path starting from the initial state and leading to the state in which the property is violated. In the following, we detail Graph Transformation Systems (GTS) that used to specify the system components and their interactions, as well as, its role in the model checking of system's behaviors.

### Graph Transformation Systems (GTS)

Graph transformation system (GTS) [64] (also called graph grammar) is a formal language for the specification of software systems, in particular those with dynamic structures. The definition of an attributed GTS consists of a triplet *(TG, HG, R)* in which $TG$ is a type graph, $HG$ is a host graph, and $R$ is a set of rules for graph transformation. $TG$ is defined by four components, $TG = (TG_N, TG_E, src, trg)$. $TG_N$ and $TG_E$ includes all node types and edge types respectively. $src$ and $trg$ are two functions $src : TG_E \rightarrow TG_N$ and $trg : TG_E \rightarrow TG_N$, that determine the source/destination nodes of an edge, respectively. The initial configuration of a system specified by GTS is presented by the host graph which is an instance of the type graph. Therefore, each component of the host graph, node or edge, must have a component type in the type graph. A host graph $HG$ may instantiate from a type graph $TG$ using a graph morphism function $type_G : HG \rightarrow TG$, in which the components of $HG$ are instantiated from $TG$. Other configurations or states of a system are generated by successive applications of transformation rules on the host graph. A transformation rule $r$ in $R$ is defined by a triplet $(LHS_r, RHS_r, NAC_r)$ in which $LHS_r$ (left-hand side) represents the preconditions of the rule whereas $RHS_r$ (right-hand side) describes the postconditions. Moreover, there may be a Negative Application Condition (NAC) for the rule $r$, meaning that the rule $r$ can be applied only when $NAC_r$ does not exist in the host graph. By applying the rule $r$ to the host graph $HG$, which is an instance model of the meta-model or type graph, a matching of the $LHS_r$ in $HG$ is replaced by $RHS_r$. Formally, a graph morphism exists between $LHS_r$ and the instance model $HG$. The application of a rule is performed in four steps: (1) find a matching of $LHS_r$ in $HG$, (2) check $NAC_r$ that forbid the presence of certain nodes and edges, (3) remove a part of the host graph that can be mapped to $LHS_r$ but not to $RHS_r$, and (4) add new nodes and edges that can be mapped to the $RHS_r$ but not to the $LHS_r$.

### GTS Applied for Program Verification

In GTS, all enabled graph transformation rules can be applied to the host (start or initial state) graph recursively. By application of the first matched rule, a new graph is generated from the host graph. This process ends when further application of any rule over the last graph becomes impossible. Hence, all reachable graphs from the host graph resulting from all possible interleavings of rule applications form a state space. Indeed, this graph-based automatic process can be leveraged to verify statically the program constraints on elements like macros and comments [65]. Moreover, researchers have proposed several program verification approaches [66–68] that model the buggy program as a graph using designed or learned meta-model. Then, they apply consecutive graph transformations to detect and locate the

bugs as well as further graph editing to fix them.

## 2.3 Chapter Summary

In this chapter, we briefly introduce the key concepts and methodologies that are related to deep learning and software quality assurance, in order to ease the understanding of the different DL testing and debugging approaches presented in this thesis.

In the following chapter, we present a literature review and discussion of the research works that exist in the area of trustworthy artificial intelligence systems.

# CHAPTER 3 Literature Review

In this chapter, we review recent empirical studies of bugs and misconceptions encountered in DNN training programs and models. We then report existing approaches for debugging and testing DNN-based software systems. Throughout the following review, we discuss the identified gaps in the literature that have been exploited in our research.

**Chapter Overview** Sections, 3.1 and 3.2, introduce studies on, respectively, DL software bugs and misconceptions when engineering DL models. Section 3.3 presents research trends in debugging DL training programs. Sections 3.4 and 3.5 describe the existing methods specialized in test input generation and OOD detection for DL models. Section 3.6 concludes the chapter.

## 3.1 Studies on DL Software Bugs

The growing application of self-learning software systems in important domains such as autonomous driving systems and facial authentication systems makes the identification and characterisation of faults that occur in such software systems of paramount importance. One of the first papers, considering faults experienced by practitioners when building DL-based programs that use mainstream data manipulation, automatic differentiation, and tensors computation libraries, is the empirical study by Zhang et al. [10]. The authors manually examined about 175 TensorFlow-based buggy programs, which were collected from Stack-Overflow(SO) Q&A posts and Github public projects, with the aim of providing insights into the main root causes and behavioral symptoms of the DL software-specific bugs. They classified DL buggy program's symptoms into crash-inducing, low effectiveness (i.e., degradation of model performance metrics), and low efficiency (i.e., degradation of program resources consumptions). The principal categories of root causes were found to be in the algorithm design like incorrect model parameter or structure (21.7%) and in the software implementation such as API misuse (18.9%) and unaligned tensor (13.7%). Then, Islam et al. [11] extended the investigated cases to include DL-based programs using other DL libraries competing along with TensorFlow, such as Theano, Caffe, Keras and PyTorch. The empirical study was conducted on a total of 2716 high-quality posts from Stack Overflow and 500 bug fix commits from Github including five popular DL libraries. Regarding causal analysis, they adopted the same list of root causes identified in [10]. However, they analyzed the prevalence of DL bugs, the fault patterns, the relationship and the evolution of different types of DL bugs. They found that data-related and logic bugs are the most severe type of DL bugs appearing

in almost half of the occurrences and major root causes of these bugs are Incorrect Model Parameter (IPS) and Structural Inefficiency (SI) showing up more than 43% of the times. As of March 2021, recently-published taxonomy of real DL software system faults [1] represent the most complete research work on the identification and classification of DL system-specific bugs. To build the taxonomy, Humbatova et al. adopted the same data sources, SO and Github, as well as the same DL libraries that have been formerly considered. Nevertheless, they conducted interviews with 20 researchers and practitioners describing the problems they have encountered in their experience. In a similar way, the validation of the final taxonomy was performed by a survey involving a different set of 21 practitioners, where 50% of the participants experienced almost all the fault categories (13/15).

In this thesis, when designing debugging approaches for DL software systems, we rely on these previous investigations of DL faults to define the scope of targeted faults, study their symptoms and their identified detection challenges, and extract reproducible and relevant faulty DL programs on which we conduct the empirical evaluations of our proposed tools.

### 3.2 Studies on DL Model Misconceptions

In the context of computer vision, Szegedy et al. [69] show that universal approximator DL models such as deep convolutional neural networks can react in unexpected and incorrect ways to even slight pixel-based perturbations of their benign inputs. Then, Engstrom et al. [70] showed the vulnerability of CNNs against craftly-transformed images using affine transformations (such as translations and rotations). In the above-mentioned categories of adversarial examples, the malicious inputs were intentionally and carefully crafted using automated data generation [71]. Instead, recent research work [72] relied on real input data and showed that CNNs cannot be robust even against simple guess-and-check of naturally-occurring situations related to the application domain, like taking pictures from another perspective angle. In more realistic conditions, Banerjee et al. [73] examined the bug reports of autonomous driving systems from a set of 12 AI-based vehicle manufacturers and found that the issues in machine learning systems and decision control represent the primary cause of 64% of all the self-driving failures (i.e., the vehicle was forced to hand over control to the driver) based on their NLP-based automated classification of issue reports that have been collected during a cumulative total of $1,116,605$ miles in California. Such findings led researchers to develop systematic adversary attacks to stress these vulnerabilities in the model learned patterns and assess the robustness of the neural networks. We refer to the extensive survey by Biggio and Roli [13] for more details about adversarial machine learning.

Although the paradigm shift of DL software development has enabled transformational

progress across a wide range of domains, DNNs are prone to emerging misconceptions, due to their structure complexity and vast number of parameters involved. Arjovsky et al. [74] discussed and illustrated how the "spurious correlations" occur in real-world scenarios, especially, when facing data biases such as selection biases, confounding factors, and other peculiarities [75–77]. Geirhos et al. [78] studied the phenomenon of "Shortcut Learning" where a statistical learning model successfully finds shortcuts that represent decision rules allowing to perform well on the available datasets but fail to transfer to more challenging and corner cases that can happen in real-world scenarios. Indeed, conventional DL pipelines involve a training phase that is characterized by a neural network design, a training dataset, and a learning algorithm. Then, an independent and identically distributed (iid) evaluation procedure that represents the testing phase, where the predictive performance of the trained DNN is estimated on unseen data samples drawn from the same distribution. Nevertheless, iid performance evaluations often fail to expose the inherent model's flaws [79], especially, when distribution drifts occur in production environments, or when a selection bias is introduced during data collection. In such cases, even iid-optimal DNNs can incorporate unjustified shortcuts and spurious associations [80,81]. Recently, a group of Google Brain researchers [79] have reported evidence of underspecification of DL pipelines in a variety of real work applications from computer vision, medical imaging, natural language processing, and clinical risk prediction. Throughout all their study cases, they show how the iid performance evaluations attribute equivalent predictive scores to multiple predictors, but the latter behave very differently and even mistakenly on unseen entries at deployment settings.

Due to their huge complexity and large numbers of parameters, modern DNNs demand intensive memory and computational requirements. To deploy them in resource-limited edge devices, numerous techniques for compressing the DNNs, such as quantization [82] and pruning [83], are designed to, respectively, reduce the arithmetic precision of tensors and remove the redundant parameters. Nonetheless, an information loss induced by such compression may lead to unstable states for the learned parameters that remain unidentified before deployment. Indeed, two DNN versions of different precision can easily achieve equivalent held-out performance on iid data, but perform quite differently in real-world settings [19, 21] because limited and often biased iid test inputs are agnostic to some relevant inductive biases encoded by the optimized DNN.

Like their ancestors in supervised machine learning, deep neural networks fundamentally inherit a closed-world assumption [84], and provide no guidance on how to handle out-of-distribution data (OOD) [85]. In fact, their data-driven learning process via empirical risk minimization (ERM) [25] can produce biased models that are susceptible to outliers, unfair to minor subsets of data, or prone to out-of-distribution samples [86]. Therefore, their

generalization capability is guaranteed on novel configurations, but drawn from the same or close distribution as the training set, called in-distribution data (ID). In reality, distributional shifts often occur in real scenarios for many reasons, such as domain transition, temporal evolution, or selection bias, which degrades the model's performance since certain captured correlations may not hold on these shifted inputs. According to the core notions of model over-parametrization and complexity, modern neural networks can behave worse than their simpler counterparts when exposed to OOD inputs [74, 87, 88].

When designing model testing approaches for DL software systems, we target essentially the issues raised by the above model misconceptions. First, we aim to systematically produce major and minor functional test suites that cover the desired requirements of the application. Second, we derive domain-specific test cases from the expert domain knowledge to assess the consistency of the inductive biases with known properties of the input-output mappings. Both proposed DL tests have the potential to enhance the conventional DL pipeline by providing task-oriented performance assessments against system specifications and deployment domain conditions. Last, we also develop an OOD detection mechanism, exploiting domain knowledge and apriori system properties, to delimit the predictive capability of the trained DNN and restrict its use on the trusted boundaries.

## 3.3 Debugging Methods Proposed for Learning Programs

In the following, we discuss the learning program debugging methods that have been proposed to check whether the ML algorithm is correctly implemented (i.e., free from coding errors) and configured (i.e., the model architecture and training hyperparameters are selected well).

### 3.3.1 Software Testing Using Pseudo-Oracle

Software testing consists in assessing the program internal states and outputs in order to find potential erroneous behaviors or bugs. This quality assessment requires a test oracle, which allows distinguishing between the correct and incorrect obtained results of the program under test. However, statistical learning programs fall into the category of "non-testable" programs [89], for which we do not have a specified test oracle, because these programs are written to determine an answer. In the field of software testing, pseudo-oracle methods, which represent partial oracles built from program properties, different implementations, etc., were leveraged to distinguish a program's correct behavior from an incorrect behavior under certain circumstances. Thus, preliminary research works on debugging ML programs have adapted the most successful pseudo-oracle testing techniques: (1) Metamorphic testing [41]

defines metamorphic relations (MRs), which relate input transformations to their expected changes on the model's predictions. For example, a metamorphic relation for testing the implementation of $\sin(x)$ can be the transformation of input $x$ to $\pi - x$ and checks the expectation on the outputs, $\sin(x) = \sin(\pi - x)$; (2) Mutation testing [45] consists of intentionally introducing faults into the program and then assessing the effectiveness of the test cases in killing the mutant program (i.e., the mutant failed the test). The ratio of killed mutants against all the created mutants is called the mutation score, which measures how much the generated test cases are efficient in detecting injected faults.

Murphy et al. [90] introduced metamorphic testing to ML in 2008. They defined several Metamorphic Relationships (MRs). For instance, they performed transformations including the addition of constant value to numerical attributes; multiplying numerical attributes by a constant value; permuting the order of inputs; reversing the order of inputs; removing a portion of inputs; adding additional instances. These MRs were shown to be effective at finding misconfigurations and bugs in three well-known ML applications: Marti-Rank, SVM-Light, and PAYL [91]. Xie et al. [92] proposed MRs specialized for testing the implementations of supervised classifiers. The MRs are based on five types of transformations: (1) application of affine transformations to input features;(2) permutation of the order of labels or features; (3) addition of uninformative and informative new features; (4) duplication of some training instances; and (5) removal of arbitrary classes or instances. The evaluation of these new MRs were able to reveal 90% of the injected faults in Weka's implementations of $k$-Nearest Neighbors (kNN) and Naive Bayesian (NB). Recent research works [93] have investigated the application of metamorphic testing to modern machine learning algorithms, with non-linear modeling capabilities, such as SVM with non-linear kernel and deep residual neural networks (ResNET). To illustrate the designed metamorphic relations, we mention some of data transformations applied to SVM such as changing features or instances orders, linear scaling of the features, and the ones adopted for ResNets, which include normalization or scaling the test data, or re-ordering the convolution operations. The evaluation was done on mutated training programs using MutPy (i.e., a tool for python code mutation) and the results show that the proposed MT approach can find 71% of the injected faults.

The above-mentioned metamorphic testing approaches have been evaluated against mutated learning programs using traditional program mutators that simply alter code instructions. This could not be representative for DL software-specific bugs. Hence, Ma et al. [94] defined a set of source-level mutation operators to mutate the source of a ML program by injecting faults. These operators allow injecting faults in the training data (using data mutation operators) and the model training source code (using program mutation operators). After the faults are injected, the ML program under test is executed, using the mutated training data or

code, to produce the resulting mutated DNNs. The data mutation operators are intended to introduce potential data-related faults that could occur during data engineering (i.e., during data collection, data cleaning, and–or data transformation). Program mutation operators' mimic implementation faults that could potentially exist in the model implementation code. These mutation operators are semantic-based and specialized for DNNs' code. Indeed, the DNN programs' mutation framework can be used to assess the effectiveness of test data and specify its weaknesses based on evaluation metrics related to the killed mutated models count. DL engineers can leverage this technique to improve data generation and increase the identification of corner-cases DNN behaviors.

### 3.3.2 Diagnosis via Visualization

Many VA systems (VAS) provide a model's diagnosis that allow detecting issues on different abstraction levels. Some of them focus on feature importance and model behaviors against real [95] or adversarial [96] examples; others focus on neuron activations [97]. Moreover, advanced visualization systems [98] [99] go beyond the diagnosis of the DNN and propose refinements required to overcome the detected issues. For instance, Liu et al. [98] constructed an interactive VAS, CNNvis, that extracts successive snapshots of the on-training CNN and analyzes it in-depth using rectangle packing, matrix ordering, and biclustering-based edge bundling in order to cluster the neurons, their interactions, their derived features and roles in relation with the target task. Instead of conducting offline diagnosis, Pezzotti et al. [99] proposed an online progressive VAS that provides continuous live feedback on the on-training DNN. Both of these previous works on diagnosis and refinement via visualization demonstrate how rich visual insights can be interpreted by an expert to identify possible modeling issues and make decisions about DNN's refinements. Nevertheless, diagnoses via VAS are interactive sessions that require DL engineers to select components to watch beforehand. This makes diagnosis via VAS an expensive process that focuses, particularly, on data and design improvements to enhance the performance results of the trained DNN.

### 3.3.3 Heuristic-based Checks on Learning Program Structure and Behavior

We have published early-stage research work [100] on cataloging the DL training pitfalls related to real DL faults. We have also proposed a learning program debugging technique that leverages heuristic-based checks to verify the program automatically during training. The list of training pitfalls were formally explained using fundamentals of statistical learning and illustrated by concrete examples. The verification routines were written based on statistical metrics estimated on multiple states aggregated from a recent bunch of training

steps. This enables debugging the DNN program on-training to uncover potential training issues such as saturated neurons' activations or highly-fluctuating loss that can be caused by widely-common DL faults. We evaluated our former approach on a mixed set of buggy training programs including mutated examples from a synthetic dataset [93] and real-world Tensorflow programs from Zhang et al. SO dataset [10]. On the same line of work, Schoop et al. [101] focused on three main common training program issues: overfitting, improper data normalization, and unconventional hyperparameters. They aimed to extend DL development tools to spot the occurrences of these issues through metric-based heuristics and produce human-readable error messages. Recently, a new debugging tool for learning programs, UMLAUT, has been implemented relying on a selection of 10 heuristics [102] that solves a subset of the common DL bugs from the DL faults taxonomy [1]. To detect issues, they statically analyze a snapshot of the program prior to the execution, or dynamically verify the program status during the training runtime. Once a heuristic check is failed, the error message is displayed to the user including context and fix suggestions summarized from the heuristic's sources by the authors. The tool was evaluated through a user study with 15 participants and the results showed its effectiveness in helping the participating developers find significantly more bugs compared to non-assisted debugging sessions. Although the codified heuristics are widely-accepted, they cover common issues mostly experienced by novice DL developers. The heuristics also make use of fixed thresholds inspired by well-spreaded practices in the ML community. This makes the approach not applicable to novel and specific learning contexts and not suitable for supporting DL research as evidenced by the evaluation that uses one and only simple neural network with injected faults. Besides, the implementation of checks was based on hard-coded callbacks built on top of Keras, which may not scale easily to support other DL libraries and even past versions of Keras. This can explain the choice of controlled experiments with junior developers and synthetic buggy Keras programs (i.e., a base code plus injected bugs) for the proposed tool's performance evaluation, than assessing its effectiveness on the real-world buggy training programs from publicly-available bug reports [1, 10, 11].

### 3.4 Model Testing Approaches

Next, we discuss the model testing methods that have been proposed to assess the performance of the learned model on systematically-produced test inputs. This builds confidence that the model behaves properly in normal functional conditions, corner-case and rare scenarios, and is sufficiently robust against exceptional or extreme conditions.

### 3.4.1 Adversarial Robustness Testing

Adversarial robustness testing consists in designing adversary data generators that allow developers to detect the data variation ($\delta$) and the natural input ($x$) yielding an adversarial example $\hat{x} = x + \delta$, for which the model does not satisfy a given property $C$. The model robustness represents the first major property that has been relatively well-studied for supervised learning problems using adversarial testing.

Szegedy et al. [69] first used box-constrained L-BFGS (i.e., Limited memory Broyden Fletcher Goldfarb Shanno algorithm) to find adversarial examples. Then, Goodfellow et al. [71] proposed Fast Gradient Sign Method (FGSM) that allows fast generation of adversarial examples, given the finding that the main cause of adversarial examples is the linear nature of neural networks. Since then, several whitebox adversarial testing methods have been released. Among them, we find Basic Iterative Method (BIM) [103] extending FGSM with iterative procedures, DeepFool) [104], crafting malicious input through iterative linearization of the on-testing neural network and other improved techniques such as Jacobian-based Saliency Map Attack (JSMA) [105] and Carlini/Wagner attack (CW) [106].

Blackbox adversarial robustness testing is less effective than whitebox ones because they require a large number of queries, but they can expose different adversarial examples and are better representative of external attack system simulations. Papernot et al. [107] explored blackbox attacks based on the phenomenon of transferability [108]. Narodytska et al. [109] performed a local-search-based attack. Chen et al. [110] and Bhagoji et al. [111] proposed blackbox attacks based on gradient estimation [112]. Moon et al [113] leveraged algorithms in combinatorial optimization. Alzantot et al. [51] recently reported about GenAttack, a gradient-free optimizer that uses Genetic Algorithms (GA) to apply imperceptible perturbations on inputs. Alzantot et al. conducted a series of experiments and reported that GenAttack can successfully fool state-of-the-art image recognition models with significantly fewer queries.

The major limitation of these adversarial robustness testing techniques is the impracticality of the generated adversarial examples. For example, the adversarial images generated for computer-vision models often contain only tiny, undetectable, and imperceptible perturbations, since any visible change would require manual inspection to ensure the correctness of the model's decision. This can result in strange aberrations or simplified representations in synthetic datasets, which in turn can have hidden knock-on effects on the performance of a ML model when unleashed in a real-world setting. These adversarial testing techniques that target only on the output of the model (ignoring the internal state details of the models under test) often fail to uncover different erroneous behaviors of the model, even after

performing a large number of tests. This is because the generated adversarial data often fail to cover the possible functional behaviors of the model adequately. An outcome that is not surprising given that the adversarial data are generated to fool the models with respect to their predictions without considering information about their structure. To help improve over these limitations, SE researchers have invented structural coverage criteria as described below, which use internal structure specificities to guide the generation of more relevant test cases.

### 3.4.2 Model-level Mutation Testing

Model-level mutation testing is a white-box testing technique that changes the parameters' values or the structure of the model under test, and then, it verifies if the test data are able to kill the mutants (i.e., there exists at least one test case that fails on the mutant). Shen et al. [114] propose five mutation operators, including (1) deleting one neuron in input layer, (2) deleting one or more hidden neurons, (3) changing one or more activation functions, (4) changing one or more bias values, and (5) changing weight value. Ma et al. [94] complement their data mutations and program mutations, by model-level mutation operators that include changing the weights, shuffling the weights between neurons in neighboring layers, etc. These operators directly change the structure and the parameters of neural network models to scale the number of resulting mutated models in an effective way, and for covering more fine-grained model-level problems.

### 3.4.3 DNN-based Structural Testing

Pei et al. proposed DeepXplore [17], the first white-box approach for systematically testing deep learning models. DeepXplore is capable of automatically identifying erroneous behaviors in deep learning models without the need of manual labeling. The technique makes use of a new metric named *Neuron Coverage (NC)*, which estimates the amount of neural network's logic explored by a set of inputs. This neuron coverage metric computes the rate of activated neurons in the neural network. It was inspired by the code coverage metrics used for traditional software systems. The approach circumvents the lack of a reference oracle, by using differential testing. DeepXplore leverages a group of similar deep neural networks that solve the same problem. The test data generation is based on domain-constrained data transformations to create many realistic visible differences (e.g., different lighting, occlusion, etc.) and automatically detect erroneous behaviors of deep neural networks against valid and realistic synthetic inputs. In the end of the testing process, the generated data is kept for future training, to have more robustness in the model.

Building on the pioneer work of Pei et al., Tian et al. proposed DeepTest [18], a tool for automated testing of DNN-driven autonomous cars. In DeepTest, Tian et al. expanded the notion of neuron coverage proposed by Pei et al. for CNNs (Convolutional Neural Networks), to other types of neural networks, including RNNs (Recurrent Neural Networks). Moreover, instead of randomly injecting perturbations in input image data, DeepTest focuses on generating realistic synthetic images by applying realistic image transformations like changing brightness, contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect, etc. They also mimic different real-world phenomena like camera lens distortions, object movements, different weather conditions, etc. They argue that generating inputs that maximize neuron coverage cannot test the robustness of trained DNN unless the inputs are likely to appear in the real-world. They provide a neuron-coverage-guided greedy search technique for efficiently finding sophisticated synthetic tests which capture different realistic image transformations that can increase neuron coverage in a self-driving car DNNs. To compensate for the lack of a reference oracle, DeepXplore used differential testing. However, DeepTest leverages metamorphic relations (MRs) to create a test oracle that allows it to identify erroneous behaviors without requiring multiple DNNs or manual labeling. Tian et al. defined metamorphic relations between the car behaviors across the proposed image-based transformations. Since it is hard to specify in advance the correct steering angle for each transformed image, they assume that the predicted angle for a transformed scene driving is correct if it varies from its genuine one to less than $\lambda$ times the mean squared error produced by the original data set. $\lambda$ is a configurable parameter that helps to strike a balance between the false positives and false negatives.

Next, Sun et al. [115] examined the effectiveness of the neuron coverage metric introduced by DeepXplore and reported that 100% neuron coverage can be easily achieved by a few test data points while missing multiple incorrect behaviors of the model. To illustrate this fact, they showed how 25 randomly selected images from the MNIST test set yield a close to 100% neuron coverage for an MNIST classifier. Thereby, they argue that testing DNNs should take into account the semantic relationships between neurons in adjacent layers in the sense that deeper layers use previous neurons' information represented by computed features and summarize them in more complex features. To propose a solution to this problem, they adapted the concept of Modified Condition/Decision Coverage (MC/DC) [116] developed by NASA. The concepts of "decision" and "condition" in the context of DNN-based systems correspond to testing the effects of first extracted less complex features, which can be seen as potential factors, on more complex features which are intermediate decisions. Consequently,

they specify each neuron in a given layer as a decision and its conditions are its connected input neurons from the previous layer. They propose a testing adequacy evaluation that is based on a set of four criteria inspired by MC/DC. As an illustration of proposed criteria, we detail their notion of *Sign-Sign(SS)* coverage, which is very close to the idea of MC/DC. Since the neurons' computed outputs are numeric continuous values, the *SS* coverage cannot catch all the interactions between neurons in successive layers. Since the changes observed on a neuron's output can be either a sign change or a value change, they added three additional coverage criteria to overcome the limitations of *SS*, i.e., Value-Sign Coverage, Sign-Value Coverage, and Value-Value Coverage. These three additional criteria allow detecting different ways in which changes in the conditions can affect the models' decision.

Despite the relative success of neuronal coverage in estimating the behavioral changing of neural networks' activations, Ma et al. [117] remarked that the real DNN state space is very large and it can be relevant to take into account the interactions between the different neurons, however, given the size of neurons, this can lead to a combinatorial explosion. To help address this issue, they proposed DeepCT, which is an adaptation of combinatorial testing (CT) techniques to deep learning models, in order to reduce the testing coverage space. CT [118] has been successfully applied to test traditional software requiring many configurable parameters. It helps to sample test input parameters from a huge original space that are likely related to undetected errors in a program. For example, the $t$-way combinatorial test set covers all the interactions involving $t$ input parameters, in a way that exposes efficiently the faults under the assumption of a proper input parameters' modeling. In DeepCT, K-way CT is adapted to allow for effectively selecting samples of neuron interactions inside different layers with the aim of decreasing the number of test cases.

Then, Ma et al. [22] generalized the concept of *neuron coverage* by proposing DeepGauge, a set of multi-granularity testing criteria for Deep Learning systems. DeepGauge measures the testing quality of test data (whether it being genuine or synthetic) in terms of its capacity to trigger both major function regions as well as the corner-case regions of DNNs (Deep Neural Networks). It separates DNNs testing coverage in two different levels. Neuron-level coverage criteria include: (1) *K-multisection Neuron Coverage (KMNC)*: the ratio of covered k-multisections of neurons; (2)*Neuron Boundary Coverage (NBC)*: the ratio of covered boundary region of neurons; (3)*Strong Neuron Activation Coverage (SNAC)*: the ratio of covered hyperactive boundary region. Layer-level coverage criteria include: (1) *Top-k Neuron Coverage (TKNC)*: the ratio of neurons in top-k hyperactived state on each layer; (2)*Bottom-k Neuron Coverage (BKNC)*: the ratio of neurons in top-k hypoactived state on each layer.

Relying on DeepGauge's fine-grained structural coverage criteria, DeepHunter [19] implements a coverage-guided fuzzing that consists in continuously applying mutations on corpus of inputs, triggering uncovered DNN's states, in order to enhance DNN's coverage and generate diverse synthetic inputs.

In our previous research work [21], we defined two levels of neuron coverage and used it to build a novel test adequacy measure that is robust against plateauing during the optimization process (i.e., reaching a state of little or no change after a time of progress). In fact, the coverage-based measure captures both local-neuron coverage (i.e., neurons covered by a generated test input that were not covered by its corresponding original input) and global-neuron coverage (i.e., neurons covered by a generated test input that were not covered by all previous test inputs). Regarding the test data generation, we have proposed, DeepEvolution, a search-based testing method specialized for DNN software that relies on population-based metaheuristics to explore the search space of semantically-preserving metamorphic transformations. Since these metaheuristics are gradient-free optimizers, they ensure the maximization of neuron coverage-based fitness, while keeping a wide variety of input transformations thanks to their flexibility (as they do not require prior assumptions).

The evaluations of all above-mentioned structural coverage-based testing methods have shown that the enhancement of structural coverage criteria are correlated with the growth of novel adversarial examples, which indicates that the unfamiliar inputs are more likely to trigger erroneous behaviors and the coverage-guided exploration promotes diversity in the generated synthetic inputs. Hence, the test data generation yields more effective test cases with both higher functionality assessment and fault-revealing ability. Nevertheless, recent research works cast doubt on the effectiveness of structural coverage. Li et al. [15] argue that the structural coverage criteria could be misleading because their preliminary exploration shows that adversarial examples are pervasively distributed in the finely divided space defined by such coverage criteria; so the correlation between high structural coverage and fault-revealing capabilities (i.e., number of exposed adversarial examples) is more likely due to the adversary-oriented search rather than the resulting enhancement of the structural coverage criteria.

### 3.4.4 Distance-based Coverage Testing

Odena and Goodfellow [119] explore the input space to find new test entries that trigger different behavioral neural network's states. They encode the activations of all neurons

as a state vector. Then, they use a fast approximate nearest neighbor algorithm based on euclidean distance to measure how much a new triggered state vector is meaningfully different from the previous ones. This allows them to decide if it should be preserved or deleted for improving the diversity of the forthcoming test input generations. Kim et al. [23] proposed a fine-grained test adequacy metric, named Surprise Adequacy (SA) that quantifies how much surprising a given input is to the DNN under test with respect to the training data. The intuition behind this criterion is that effective test inputs should be sufficiently surprising compared to the training data. The surprise of an input is quantitatively measured as behavioral differences observed in a given input relatively to the training data. Kim et al. defined two concrete instances of their SA metric, given DNN's activations trace (AT) that represent a vector of neurons' activations : (1) Likelihood-based SA which uses Kernel Density Estimation (KDE) to estimate the probability density of each activation in AT, and computes the relative likelihood of new input's activation values with respect to estimated densities; and (2) Distance-based SA which uses the Euclidean distance between a given input's AT and the nearest AT of training data in the same class.

### 3.5 OOD detection strategies

A wide range of detection strategies [120, 121] has been released to overcome the out-of-distribution challenge. Deep generative models [122–124] have been leveraged to model efficiently the distribution of inputs $p(x)$ on the training samples. Then, a membership test is performed on any input $x$: if $p(x)$ is low, $x$ will be assigned to $\in \mathcal{D}_{ood}$, and vice versa. However, this generative modeling faces limitations on large-scale and/or complex input distributions. In contrast, model-dependent OOD detection methods focus on the distribution of model's intricacies, such as hidden features, softmax probability outputs, and uncertainty scores, rather than directly modeling input distributions [125]. Several model-dependent methods [85, 121] have been proposed for classification neural networks. A few research works have focused on regression neural networks through thresholding on uncertainty estimates [126]. Bayesian approaches [127–129] approximate the posterior distribution of neural networks parameters through an ensemble of models. Regarding Non-Bayesian ensembling approaches, Kendall and Gal [130] adds the expected prediction's variance as additional network's output. Hence, the variance of a prediction is also improved by the learning algrithm, guided by the minimization of the negative log likelihood on the data. Mi et al. [131] proposed training-free uncertainty estimation that leverages the average of output deviations under input or feature map perturbations as a surrogate for uncertainty measurement. One challenge with these uncertainty-based methods is their reliability degeneration

on out of distribution inputs on which they may be falsely overconfident [132]. Therefore, OOD examples can be identified by their predicted variances beyond the confidence interval obtained for ID samples. Mi et al. [131] proposed training-free uncertainty estimation that injects tolerable perturbations into either inputs or feature maps during inference and uses the induced variance of the outputs as a surrogate for uncertainty measurement. Recently, Santana el al. [133] proposed a classification-based OOD detector to separate benign and adversarial inputs, relying on multiple features extracted from the inputs in order to build robust DL regression models that forecast photovoltaic power generation. In this thesis, we aim to exploit apriori system properties like smoothness in order to mitigate the flaws of data-driven uncertainty estimations, and to develop a novel OOD detection strategy for regression models under conditions of smoothness that makes use of the mapping function's sensitivity profiling to capture potentially OODs.

## 3.6 Chapter Summary

Over the last decade, researchers and big tech companies have been working increasingly to investigate the reliability issues raised by DL-enabled software systems and to develop quality assurance approaches to ensure their trustworthiness when deployed in real-world settings. In this chapter, we explain the main DL program bugs and model misconceptions reported in the literature. We then review the debugging and testing techniques proposed to detect these issues at both the implementation and modeling levels. We also review out-of-distribution detection mechanisms designed to detect when a trained model has crossed the trusted boundary. Finally, we present the identified gaps that we have exploited to release our novel techniques described in the following chapters.

# CHAPTER 4 NEURALINT: A STATIC RULE-BASED DL PROGRAM DEBUGGER

Many developers, entrepreneurs, and researchers are showing an increasing enthusiasm in developing Deep Learning (DL) applications in a variety of domains. Easy-to-use libraries such as TensorFlow or Keras have been released to simplify the development process. However, leveraging these libraries to implement a training program is still challenging, in particular for developers who are not experts in machine learning and neural networks. Indeed, DL developers usually relies on the rich set of APIs provided by DL libraries that can be configured and adapted to solve new problems. However, their generic-purpose nature requires the incorporation of many assumptions. Their misuses, i.e., the usage of APIs without fully understanding their inner functioning and–or checking that their assumptions are fulfilled, are likely to result in erroneous behavior. In addition, poor design choices and coding mistakes can also result in divergences between the written source code and the algorithm's mathematical formulation of the implemented training program. For instance, let us consider the program in Figure 4.1 which is extracted from Stack Overflow (SO) post #44322611 and is reported to have a low accuracy.

```
#train data
data1 = DataFetch('orange', ...)
data1 = DataFetch('apple', ...)
...
#one-hot encode outputs
y_train = np_utils.to_categorical(y_train)
#number of classes is 2: {orange, apple}
number_classes = y_train.shape[1]
#create the model
model = Sequential()
model.add(Conv2D(224, (11, 11), ...))
model.add(Dropout(0.2))
model.add(Conv2D(55, (5, 5), ...))
model.add(MaxPooling2D(pool_size(2, 2)))
model.add(Conv2D(13, (3, 3), ...))
model.add(Dropout(0.5))
...
model.add(Dense(num_classes), activation='softmax')
# compile model
model.compile(loss='binary_crossentropy', optimizer=SGD, ...)
```

Figure 4.1 Simplified example DL program from SO_44322611.

This program which implements a Convolutional Neural Network (CNN) has three issues. The first issue (i.e., ①) is a bug due to the incompatibility between the *softmax* as out-

put activation and *binary_crossentropy* as loss function. In fact, developers should have chosen *categorical_crossentropy* because it works with one-hot encoding labels and softmax activation to solve multi-label classification problems (including binary labels problems)[1]. This kind of API misuse error can be identified by verifying the consistency of the involved mathematical operations. For instance, error ① induces an inconsistency between the loss function and the last layer activation. The other two errors are ② decreasing filters count $224 > 55 > 13$ and ③ decreasing filtering spatial size $(11, 11) > (5, 5) > (3, 3)$, which represent poor structural CNN choices that violate the common design patterns of effective and optimal CNN architectures [135, 136]. These structural errors are often detected through manual code reviews which is time consuming. A static code analysis using automated tools can significantly speed up this process. In this chapter, we examine common structural errors and design inefficiencies occurring in DL programs and propose *NeuraLint*, a model-based verification approach for their detection. To design *NeuraLint*, we first propose a meta-model for DL programs that includes their base skeleton and fundamental properties. This meta-model captures their essential properties independent of available DL libraries. Considering the proposed meta-model, we specify for each fault or design issue, a verification rule that can be used to detect its occurrence. Finally, we propose a checking process to verify models of DL programs that are conforming to the meta-model. We employ graph transformations to implement *NeuraLint*. We present a type graph for the meta-model and graph transformations for the verification rules. We evaluate our approach *NeuraLint* by finding various types of faults and design issues in 28 synthesized examples built from common problems reported in the literature [1] and 34 real-world DL programs extracted from GitHub repositories and SO posts. The results show that *NeuraLint* effectively detects faults and design issues in both synthesized and real-world examples.

**Chapter Overview.** Section 4.1 presents our meta-modeling of DL programs. Section 4.2 introduces the studied issues and rules for their detection. Section 4.3 presents our proposed approach *NeuraLint*. Section 4.4 reports the empirical evaluation of *NeuraLint*. Finally, Section 4.5 concludes the chapter.

## 4.1 Meta-modeling DL Programs

With the proliferation of libraries supporting the development of DL programs, a fundamental question emerges: is there any generic representation of DL programs that is independent from these libraries? In other terms, can we define a meta-model of DL programs and how

[1] We refer the reader to [134] for more practical use cases about how to choose the last layer activation and loss function when using the Keras Library.

can we model a DL program? Answering this question would pave the way for the application of model-driven engineering techniques to the detection of errors in DL programs. In [137], researchers proposed a meta-model for meta-learning. They presented an overview of the meta-learning concepts –on a meta-modelling level– with possible variabilities and discussed how their meta-model could be integrated into existing modelling frameworks and tools. However, while their meta-model includes "Learning Block", "Learning Algorithm", "Optimizer" and "Hyperparameters", no further details like specifications of learning algorithms or blocks are presented and they did not explore the possibility of identifying errors in machine learning models. In this section, we present a particular meta-model for DL programs and our approach for meta-modeling of such programs to perform static analysis of DL programs. We describe possible variabilities of the meta-model and how concrete DL programs can be generated from it. We think that a generic meta-model for DL programs can significantly facilitate the use of DL in various applications and would be helpful for understanding DL programs written by developers using third-party DL libraries. In fact, model-driven engineering is a perfect tool to make this idea come to life and ease the process of developing and debugging DL programs.

### 4.1.1 A Meta-Model for Deep Learning Programs

A DL program has different components. The core of each DL program is a DNN. For the sake of simplicity, we only consider the feedforward multilayer perceptron (MLP) architecture. Like other computational models, DNN attempts to find a mathematical mapping from the input into the output during a learning phase. Usually, a set of inputs and desired outputs (or targets) is provided for learning, which is called Dataset. Therefore, our meta-model includes three main parts: Architecture of DNN, Learner, and Data. Since we have used GTS for modeling DL programs, our proposed meta-model is represented by a type graph. The proposed type graph is illustrated in Figure 4.2. The node representing the **DL program** has three edges to **Architecture**, **Learner** and **Data** nodes indicating its main components. In the following, we describe the meta-model in detail. It should be noted that our aim of meta-modeling is the detection of faults in DL programs; therefore the most relevant components have been incorporated into the meta-model.

#### Architecture of deep neural network

An architecture starts with the input layer, continues with some hidden layers and ends with the output layer. We have considered a distinctive node for the **InputLayer** because of its importance but all other successive layers are modelled as **Layer**. Each layer has a **size**

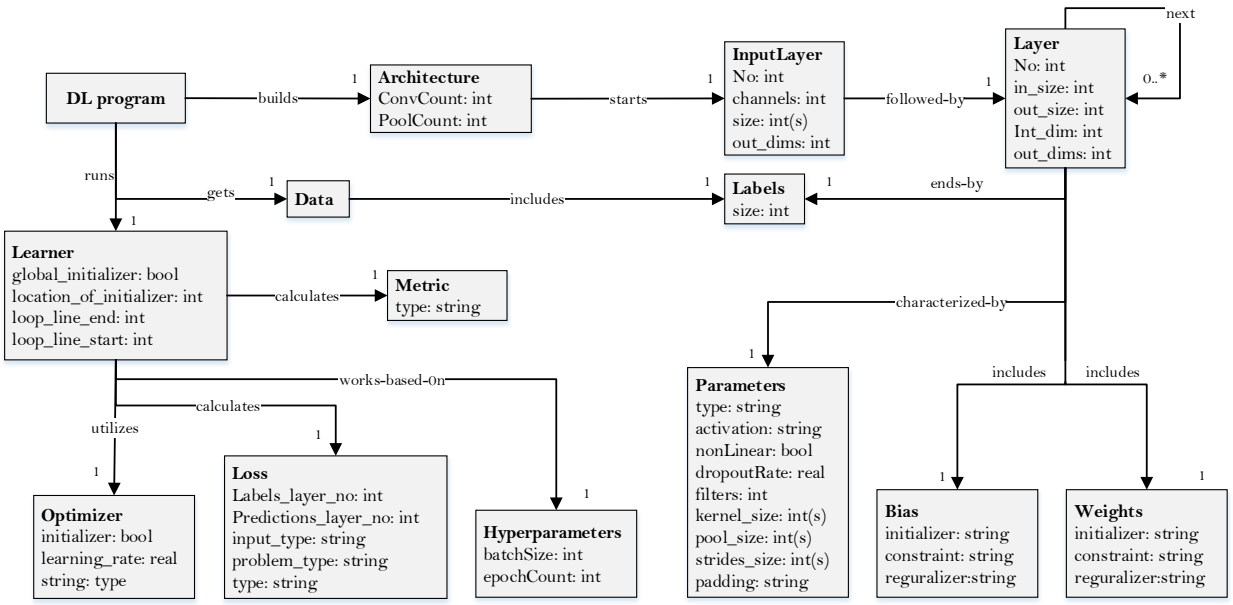



Figure 4.2 The proposed meta-model (type graph) for DL programs.

indicating the number of neurons in that layer. There are specific properties among nodes that are modelled as edges. For example, **Architecture** starts by **Input Layer**, **Input Layer** is followed by other **Layer**s, each **Layer** may have next layers and each **Layer** has a **Type** as an attribute. There are different types for a layer in DL, e.g., dense, 1D and 2D convolution, pooling or data processing layers like flatten. There may be other attributes for a layer like **Bias**, **Weights**. An architecture ends with **Labels**, the desired outputs of DNN that are used to calculate the error of the network in **Loss** function. Actually, **Labels** is a part of **Data** associated with the DL program.

### Learning algorithm

A DL program normally employs a learning algorithm, **Learner** in our meta-model, to learn the mapping from inputs to outputs. A **Loss** function is used to calculate the error of a neural network in matching the target (desired) and output value during training. The goal of learning is minimizing the loss by modification of the network's parameters (weights) by an **Optimizer**, e.g., Adam or stochastic gradient descent (SGD). The overall performance of the network is measured using a **Metric**. Moreover, there are some **Hyperparameters** like the number of epochs or batch size.

**Data**

This node contains **Labels**, meta-data, features, and related information about the data set, like shuffling and batching.

### 4.1.2 DL programs modeling

A model of a DL program includes components that form its source code. There are two ways to build a model of DL programs: configure an arbitrary model directly or transform a DL program to a model. One may design a model for a DL program that conforms to the proposed meta-model by configuring each component of the meta-model. Starting from an empty model, layers are added one by one to **Architecture**; making a chain of layers that starts by **InputLayer** node, follows by other **Layer** and ends with **output**. Each layer is configured separately to set **Parameters**, **Weights**, etc, and once a layer is configured completely, the model will proceed to the next layer. Other components of DL programs like **Learner** and **Data** are configured respectively. This process is similar to what a developer does when developing a DL program using popular DL frameworks. Therefore, the meta-model and resulting models would be realistic from a practitioner point-of-view and sufficiently flexible in representing plenty of DL programs.

On the other hand, a model could be configured according to a DL program that has already been developed by a programmer. The source code of a DL program is converted to a model, which is an attributed graph. Dedicated convertors are programmed in *NeuraLint* to convert a DL program written by different DL libraries to its model. The source code of a program is parsed to extract relevant information that is necessary to configure the model. The meta-model is generic enough to be independent of any specific DL library. Hence, we can have a model of a DL program that conforms to the meta-model; making possible further investigations on the model, such as verification. Apart from the work and analysis that are presented in the rest of this paper, we believe that this meta-model can be very useful to understand DL programs written by third parties. It will be helpful in understanding the development activities of DL practitioners; the way they write DL programs and the type of faults that they experience.

## 4.2 Model-based Verification Rules

In this section, we present the proposed rules for detecting faults in DL programs. First, we report the adopted methodology for extracting rules. The rules are then described in Subsection 4.2.2. Afterward, we present a discussion on the application scope of rules. Finally,

we describe the approach adopted to implement the rules using graph transformations.

### 4.2.1 Methodology

Figure 4.3 illustrates the adopted methodology for extracting the rules. We have explored three main sources: datasets of buggy DL programs (including bug repairs), relevant research papers and official DL libraries' tutorials. In the first step, we manually inspected the labeled bugs in the public datasets of buggy DL programs released by former research studies [1, 11, 138] collected from SO and GitHub with the objective of discovering finer root causes of bugs. Zhang et al. [138] published the first empirical study on real-world DL bugs occurring in Tensorflow-based software systems including their high-level root causes and symptoms. Then, Islam et al. [11] extended the investigated cases to include DL software systems written using other competitive DL libraries such as Pytorch and Caffe, and studied the relationship and the evolution of different DL bug types. Last, Humbatova et al. [1] refined the former bug investigation [11, 138] into a taxonomy of real faults that occur in DL software systems. In the second step, we have inspected bug fixes suggested in accepted answers of SO posts and fix patterns adopted in GitHub samples to identify patterns followed to fix the reported bugs.

We reviewed research studies on DL bugs [1, 11, 138] and fundamental DL design principles [135, 136, 139, 140] in the third step. Regarding the former, the aim was finding rules for validating the correctness of model structure and configuration choices through the DL program's model drawn from the code. We build on these previous works to specify rules that can be used to detect occurrences of different types of issues in DL programs and validate the conformity of the DNN design to common patterns through static code inspection. However, most of the DNN design patterns and principles have been deduced from state-of-the-art CNN architectures [141, 142] that have shown their effectiveness on public computer vision datasets and competitions such as ImageNet classification [143] or COCO object detection [144]. Thus, we aim to report warnings to the user whenever a poor design choice is spotted with respect to these empirical research studies on DNN design principles. This would likely steer the user to redesign his model in order to avoid the performance degradation either at the training or at the inference mode. Finally, to support practitioners in their DL program debugging, we also proceed with a dual analysis over the commonly reported APIM (API Misuse) bugs and the official DL libraries' tutorials.

In the end, we have come up with 23 rules for detecting bugs and issues in DL programs. The rules are organized into different high-level root causes as initially introduced in [138], namely Incorrect Model Parameter or Structure (IPS), Unaligned Tensor (UT), API Misuse (APIM), and Structure Inefficiency (SI).

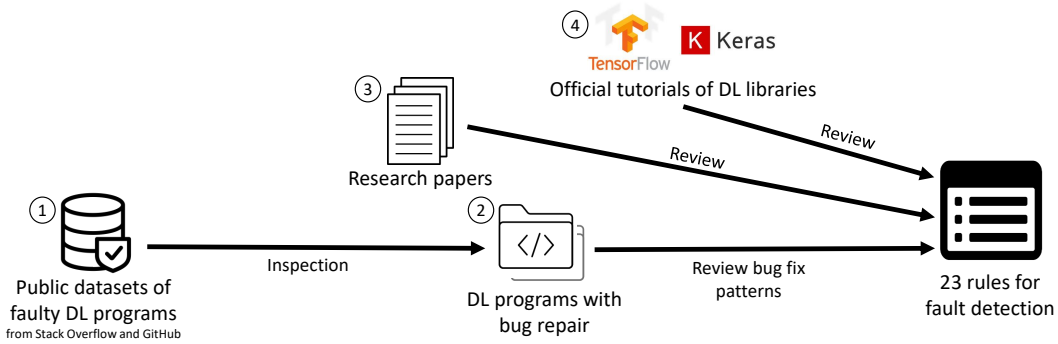



Figure 4.3 The adopted methodology for extracting rules from different sources.

### 4.2.2 Rules

#### Incorrect Model Parameter or Structure (IPS)

IPS bugs are related to modeling faults that arise from either an inadequate model parameter like learning rate or an incorrect model structure like missing or redundant layers [138]. The major symptom of IPS bugs is anomalous training behaviors leading to low effectiveness such as low precision and a huge loss.

**Rule 1: Asymmetric Units Initialization.** *The initialization of weights should not be constant to break the symmetry between neurons* [139]. For instance, a common mistake is to start with null weights, which eliminates asymmetry between the neurons (i.e., all the neurons would output the same value, and then, would receive the same gradients).

**Rule 2: Null Biases Initialization.** *The initialization of biases is preferred to be zeros* [139]. It is a common practice to expect that the outputs could be totally explained by the input features. Indeed, no custom initial bias provided a consistent improvement, but it may weaken the learning.

**Rule 3: Non-Linear Activation Requirement.** *Activations for learning layers (i.e., convolution and fully-connected layer) should be a non-linear function.* This key attribute is needed to enhance the ability of DNN to model highly nonlinear mappings and draw complex shape decision boundaries [140].

**Rule 4: Unnecessary Activation Removal.** *Multiple and redundant connected activations are not allowed.* Since all activation functions are designed to transform real values into a restricted interval [140], successive activations applied to the same features can make their last activation unable to produce its full output range.

**Rule 5: Class Probability Conversion.** *A last layer activation is required to transform the logits into probabilities for classification problems.* In detail, sigmoid ($\sigma(z) = \frac{1}{1+e^{-z}}$) and

softmax ($\sigma(z)_i = \frac{e^{-z_i}}{\sum_{j=1}^{L} e^{-z_j}}$, for $i = 1, ..., L$ and $z \in \mathbb{R}^L$) are, respectively, needed to normalize outputs with single unit ($z \in \mathbb{R}$) and multiple units ($z \in \mathbb{R}^L$). A common mistake is to use softmax instead of sigmoid for binary classification without one-hot encoding beforehand, which totally obstructs the learning because the outcome is always equal to $1 = e^{-z_1}/e^{-z_1}$, $z \in \mathbb{R}^1$.

Indeed, the above-mentioned rules show that static DL code analysis can help detect earlier structural bugs and misconfigurations, however, incorrect parameters like learning rate or neural network size (width and depth) could be identified through empirical evaluation of the model on the underlying data.

**Unaligned Tensor (UT)**

The computational units in a DNN graph are mostly tensor-based operations, where each one receives and returns tensors (i.e., multi-dimensional arrays). Their connections can hide issues related to the compatibility of tensors' shapes. DL developers often fail to express and manipulate the shapes of tensors properly [39] because DL libraries mask all the algebra computations and dynamic shapes' inference details. A bug triggered during the DNN graph construction when the shapes of one operation's tensors do not match is called an Unaligned Tensor (UT) [138]. The major symptom of UT bugs is runtime errors because the underlying tensor-based operation could not run on two incompatible tensors. However, the dynamic shape inference included in most DL libraries often makes the exception of incompatible shapes triggering far from its localization in the DL code; so the error message can be misleading. In the following, we describe various DNN layers' connectivity and configuration rules that can be checked on the DL model to identify the UT bug type and localization.

**Rule 6: Consecutive Layers Compatibility.** *A processing layer that operates on a N-dimensional tensor, should receive a valid input tensor with exactly N-dimensional shape.* For instance, a Conv2D layer works on 4-D tensors, i.e., $[samples, height, width, channels]$, but a Dense layer works on 2-D, i.e., $[samples, units]$, which means a reshape layer is needed to flatten the convolutional feature space before starting the dense layers' inference.
**Rule 7: Spatial Size Agreement.** *A processing layer should receive sufficient-sized feature space to perform its spatial filtering or pooling.* For instance, 2-D processing layers like Conv2D and MaxPooling2D require a size of feature space greater or equal to their local window size, i.e., $(window\_height \leq input\_height)$ and $(window\_width \leq input\_width)$.
**Rule 8: Reshaped Data Retention.** *A reshape layer should preserve the total data ele-*

*ments.* More specifically, we verify that the product of original tensor dimensions equals the product of reshaped tensor dimensions.

**Rule 9: Separate Item Preservation.** *A reshape layer should never alter the size of elements (i.e., first dimension).* Otherwise, the reshape would provoke an overlap between data items (i.e., points in the feature space), and as a consequence, invalidate the following layers designed to process each data item, independently.

### API Misuse (APIM)

APIM bugs are the ones introduced by practitioners who misunderstand some essential assumptions made by the used DL APIs [138]. Indeed, most DL libraries encode the DNN as an acyclic computational graph where the edges are tensors and the nodes correspond to operations. The operations include all the supported computational units that form the linear computations, activations, gradient estimations, etc. Programmatically, practitioners describe their designed DL program by inserting and configuring built-in DL routines, and connecting them by putting the outputs of one operation as inputs to another. When these routines are added without fulfilling their usage conditions or without context alignment, the DL program would not reflect the designed DL model or cannot be successfully executed by the DL core framework, which leads, respectively, to low effectiveness or runtime exceptions. Below, we detail the verification rules that should be executed on the generated static analysis-based graph model to confirm the existence of essential DL program's components and their consistency with API assumptions and recognized application context.

**Rule 10: Valid Loss Linkage.** *The loss should be correctly defined and connected to the last layer activation in accordance with its input conditions (i.e., shape and type).* For instance, the input type for cross-entropy based losses could be either logits or probabilities. Indeed, numerically stable implementations regarding the cross-entropy based losses require merging both loss and last activation functions together to rewrite the join formula carefully without any risk of $\log(0)$ or $\exp(\infty)$. However, ignoring this difference between theoretical loss functions and their numerically-stable implementations gives rise to a common mistake in the development of DNN programs, as passing activated output to this logit-based loss would cause redundant activations.

**Rule 11: Valid Optimizer Linkage.** *The optimizer should be correctly defined and connected to the computational graph.* Depending on the DL library, it could be either connected to the loss (e.g., TensorFlow) or the learnable parameters (e.g., Pytorch).

**Rule 12: Single Global Initialization.** *The learnable parameters should be totally initialized once at the beginning of the training.* For some DL libraries (e.g., TensorFlow), this

mandatory condition should be carried out by the developer.

**Rule 13: Zero Gradients Reset.** *The gradients should be re-initialized after each training iteration.* This clears old gradients from the last step; otherwise accumulating the gradients hinders the optimization process. Some DL libraries (e.g., Pytorch) delegate this necessary reset step to their users.

**Rule 14: Iterative Training Procedure.** *The loss minimization problem should be solved iteratively with continuous update of parameters.* Depending on the granularity level of the API used, it could be a native loop of optimization routine calls or a single call of a configurable fit function.

### Structure Inefficiency (SI)

SI issues reflect a misconfiguration in the DNN design and its structure that leads likely to performance problems, contrary to IPS bugs that leads to functional incorrectness [138]. SI issues may result in performance inefficiencies (like long time of model training/inference) or poor predictions (like low classification accuracy). As an example, large feature-maps, especially in the early layers, provide more valuable information for the CNN to utilize and improve its discriminative power. Therefore, it is crucial to avoid prematurely down-sampling and excessive appliances of pooling. Otherwise, the model will lose some information extracted in early layers resulting in poor performance. Since the best trained model cannot guarantee 100% of accuracy, it is challenging to detect design issues by assessing the performance of the obtained models. Indeed, some misconfigurations and poor design choices may definitely introduce inefficiencies on the internal functioning of the DNN or one of its components, which can hinder the expressiveness of mapping functions, memory and compute consumption. For example, when increasing the depth of a DNN, it is important to control both the model size and the computational cost (regarding the specific task); otherwise, stacking a high number of layers can worsen the performance.

**Rule 15: Effective Neurons Suspension.** *The dropout layer must be placed after the maximum pooling layer to be more effective.* Considering the case studies with max-pooling layers [32], the dropout has been applied on the pooled feature maps, which becomes a heuristic followed by the state-of-the-art CNN architectures [141, 142]. The intuitive explanation is that dropping out the activation before the pooling could have no effect except in cases where the masked units correspond to maximums within input pooling windows because the max-pooling would keep only these maximums as inputs for next layers.

**Rule 16: Useless Bias Removal.** *A learning layer should no longer include a bias when it is followed by batchnorm.* Batchnorm applies, after the normalization, a linear transforma-

tion to scale and shift the normalized activations $\hat{a}_i = \alpha a_i + \beta$, where $\alpha$ and $\beta$ are learnable parameters. This allows DNN to compensate for any loss of information by the value distortions in order to preserve its expressive power. Since, batchnorm already adds a $\beta$ term fulfilling the same role of bias, so "its effect will be canceled" [34] in the presence of a bias.

**Rule 17: Representative Statistics Estimation.** *Batchnorm layer should be before the dropout.* Otherwise, batchnorm computes non-representative global statistics (i.e., moving average and moving variance) on the dropped outputs of the layer. Li et al. [145] discussed the reason behind this disharmony between dropout and batchnorm and showed experimental results reinforcing their explanation.

**Rule 18: Pyramid-shaped Construction.** *The area of feature maps and the width of fully-connected units should be progressively decreasing over the layers.* It has been shown [146] that the progressive size reduction of activations implicitly forces the neural network to find and learn more robust features. Hence, it significantly improves its predictions, since the network decisions are based on more discriminative and less noisy features.

**Rule 19: Maximum Pooling Domination.** *Max-pooling is the preferred down-sampling strategy.* In fact, down-sampling [147] can be done by max- or average-pooling or strided convolution (strides greater than 1). Nevertheless, max-pooling operation has been shown [148] to be extremely superior for capturing invariances in data with spatial information, compared to other downsampling operations.

**Rule 20: Gradual Feature Expansion.** *The number of feature maps should be gradually expanded while the feature map area is retracted.* The growth of feature maps count is recommended [149] to compensate for the loss of representational expressiveness caused by the continuous decreasing of the spatial resolution of the learned feature maps. Throughout the layers, the feature space becomes synchronously narrower and deeper until it gets ready to be flatten and fed as input vector to the dense layers.

**Rule 21: Local Correlation Preservation.** *The local window size for spatial filtering should generally increase or stay the same throughout the convolutional layers.* It makes sense that by using CNNs, the locality of information is crucial for performing the task. Thus, it is important to preserve locality throughout CNN to guarantee its success in detecting various features and relations between them [150]. Furthermore, early convolutional layers learn lower level features while deeper ones learn more high-level and domain specific concepts. It is recommended [151, 152] to start with small spatial filtering to collect much local information and then gradually increase it to represent more compound information.

**Rule 22: Maximum Information Utilization.** *Deep CNN should not apply pooling after every convolution.* For instance, we use, as approximations, the minimum of 10 layers to consider a CNN deep and 1/3 as threshold for the proportion of pooling layers with respect

to the total of convolutional layers (convolution + pooling) to pinpoint a high amount of pooling. In fact, it has been shown [152–154] that larger feature-maps, especially in the early layers, provide more valuable information for the CNN to utilize and improve its discriminative power. Therefore, it is crucial to avoid prematurely down-sampling and excessive appliance of pooling.

**Rule 23: Strive for Symmetry and Homogeneity.** *Deep CNN should favor blocks of* 2*,* 3 *or even* 4 *homogeneous convolutional layers with similar characteristics.* Indeed, going deeper does not refer to simply maintaining stacking a series of convolution and pooling layers. Advanced CNN architectures [155–157] have shown the benefit of having several homogeneous groups of layers, where each one is specialized to achieve a particular goal. Indeed, building blocks of convolutional layers with similar characteristics (i.e., the same number of feature maps and feature map sizes) increases the homogeneity and the structure symmetry within the CNN. Hence, larger kernels can be replaced into a cascade of smaller ones, which enhances the nonlinearity and yields better accuracy [151]. For instance, one $5 \times 5$ can be replaced by two $3 \times 3$ or four $2 \times 2$ kernels. Moreover, spatial filtering with reduced size decreases massively the computation power requirement because recent NVIDIA cuDNN library (version 5.x or higher) is not optimized for larger kernels such as $5 \times 5$ and $7 \times 7$, whereas CNN [151] with entirely $3 \times 3$ filters achieved a substantial boost in cuDNN performance.

### 4.2.3 Application scope

The rules are defined to support the debugging of DL programs through static analysis-based graph models. On the first hand, we have been limited to the information that could be parsed from the source code of a DL program. For instance, there are some model parameters that should be experimentally tested to assess their adequacy for the underlying problem, particularly for IPS bugs. Thus, the bugs related to data (type, format, and preprocessing steps) and hardware issues (GPU configuration and required memory) are excluded from the rules and debugging scope because the information needed to diagnose the issue and identify those bugs are mostly out of the static DL code scope. In fact, these types of bugs could be better detected using Python and GPU firmware native debugging tools that help inspect step by step the executed statements at runtime. On the other hand, we have defined a high-level meta-model that could be instantiated to represent any DL program; so, bugs and intricacies that are related to specific DL libraries or APIs are discarded from the verification routines. Referring to the identified high-level root causes of DL bugs [138], we did not consider API Change (APIC), which reflects anomalies by a DL program upon a new release of the used library and Confusion with Computation Model (CCM), which includes bugs

arising from misunderstanding the DL library computation model such as DAG and deferred execution of Tensorflow, and bugs which are related to regular programming mistakes like for any traditional software.
According to the categories in the most recent taxonomy of DL faults [1], we mention the type of faults that could be covered by the proposed rules: *Wrong Tensor Shape, Wrong Shape of Input Data, Model Properties, Layers, and Loss Function*. Based on the count of manually-analyzed real-world buggy programs in [1], we found that the covered DL bugs/issues in *NeuraLint* represents 51.7%, of all reported DL buggy samples in the taxonomy. In the following, we report a prevalence ratio for each type of bugs, i.e., the number of buggy DL programs assigned to the underlying category divided by the total number of buggy DL programs in [1]:

- *Wrong Tensor Shape (*14.1%*).* It refers to errors leading to unexpected tensor shape and mismatch between operations' shapes of tensors.
- *Wrong Shape of Input Data (*14.8%*).* It assembles the bugs caused by invalid shapes of input data for a computational layer including input layer, hidden layers, and output layer, as well as the shape of math function's inputs.
- *Model Properties (*2.7%*).* It comprises improper modeling choices that can dramatically degrade the DNN's performance such as missing, wrong model initialization, or sub-optimal model structure.
- *Layers (*15.4%*).* It contains the bugs related to layers including missing, redundant, misconfigured and wrong neural network layers.
- *Loss Function (*4.7%*).* It covers different issues in relation with the loss component such as missing, inadequate and wrong loss function.

### 4.2.4 Representing Rules as Graph Transformations

In this chapter, the meta-model is presented as a type graph and each model is a graph, instantiating the type graph. Each DL program is converted to a graph, as well. As a straightforward approach, graph transformations are chosen to implement the verification rules. Each verification rule is implemented as one or some graph transformations or graph processing operators. In fact, graph transformations are used to detect possible faults in a model, faults that are caused by violating the verification rules. Consequently, a transformation is applicable where the conditions of the corresponding rule are violated. In other words, if conditions of a verification rule are violated representing a fault in a model then the graph

operation(s) of that rule will be applicable. Graph transformations are very flexible to find violation of some conditions in a graph. Recalling that a graph transformation $r$ is defined by a triplet of $(LHS_r, RHS_r, NAC_r)$, a specific condition would be checked by finding a match of $LHS_r$ in the graph and/or the absence of $NAC_r$. Once a graph operation is applied, i.e., detecting a fault in a part of the graph, a specific fault code is added to the node or edge in which the violation occurred. This action is represented by the right hand-side of the rule $RHS_r$.

Figure 4.4 illustrates one of verification rules implemented as a graph transformation, showing LHS, RHS, and NAC. The transformation is an implementation of Rule 4 which asserts that: *Multiple and redundant connected activations are not allowed*. Developers usually add activations after learning layers (like convolution and dense layers) to produce proper output signals. LHS shows a learning layer with the **type** of 'dense', 'conv1d', 'conv2d' or 'conv3d' in its **Parameters** node, followed by two consecutive layers containing the **type** of 'activator' and the **nonLinear** as *True* in their parameters. A positive closure is used on the label of incoming edges to activation layers (*next+*). This states that activations may appear in any **Layer** node on the path beginning from the learning layer and including multiple *next* edges ($\geq 1$). To be sure that another learning layer would not appear on this path, e.g., false detection of the next learning layer followed by its only activation, NAC forbids the existence of any learning **Layer** node on the subpath leading to activation nodes. If such a match is found in a model (graph of DL program), Rule 8 is violated. Therefore, RHS just adds a **Faults** node with relevant fault code to the faulty component, i.e., learning layer. Because of space limitation, we cannot present in the paper all the graph transformations implemented for our model verification. We refer interested readers to the source code of *NeuraLint* which is available online [158].

## 4.3 A Model-based Verification Approach for DL programs

In this section, we describe our approach, *NeuraLint* for detecting faults in DL programs. *NeuraLint* is a model-based automated approach that performs a static analysis of a DL program to detect faults and design inefficiencies. Algorithm 1 shows the pseudocode of *NeuraLint*. The inputs are a DL program and a graph grammar, i.e., a set of graph transformations rules. As presented in Algorithm 1, *NeuraLint* has three main steps: extract a graph from the DL program, perform graph checking and generate a report from the resultant graph. At first, the DL program is modeled as a graph that conforms to the proposed meta-model, i.e., type graph. Then, a checking process runs to find bugs/issues in the model. This process attempts to apply rules to the graph and stops when further rule application

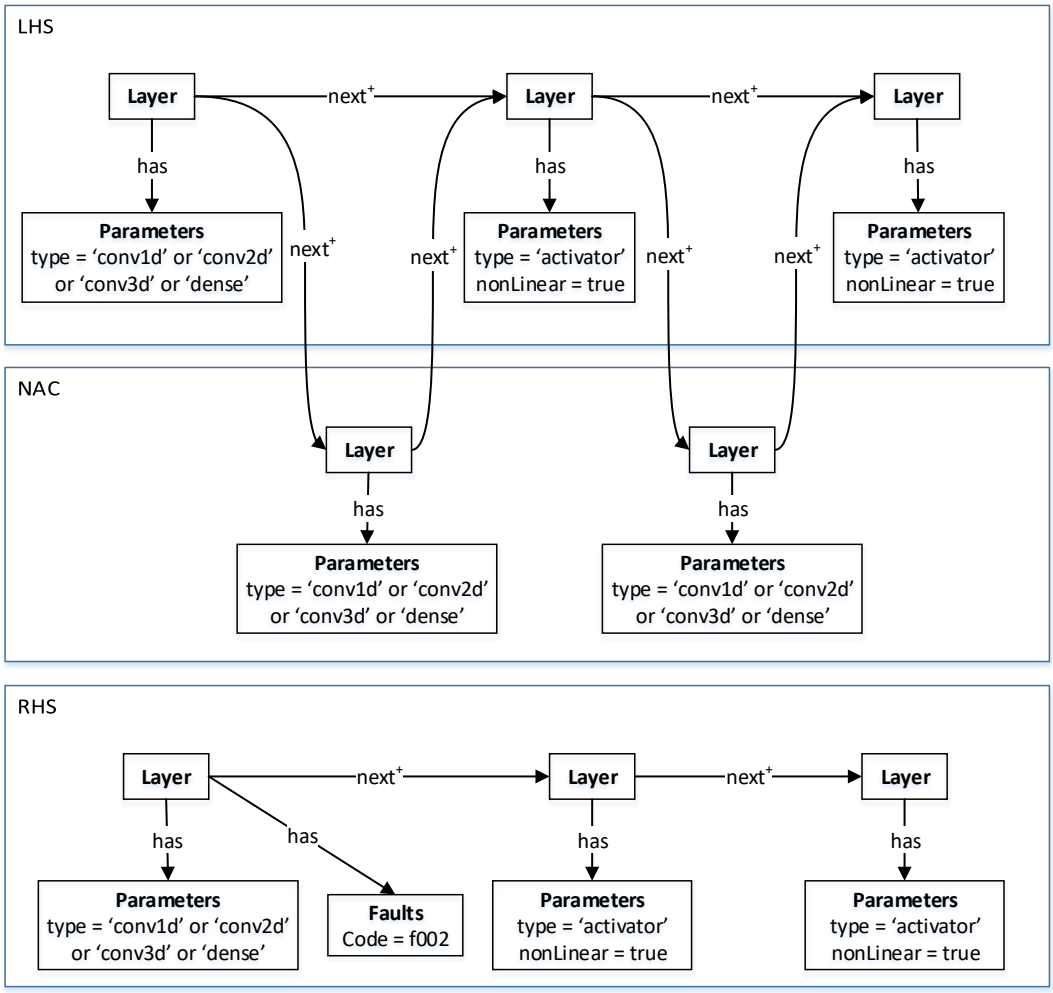



Figure 4.4 An Example of Graph Transformation Rules: Implementation of Rule 4.

becomes impossible. Then, *NeuraLint* traverses this graph to generate a report for the user, containing a description of the faults and design issues found for each component. Except graph checking and graph transformations, all other parts of *NeuraLint* are implemented in Python. We discuss details of each step in the rest of this section. The source code, developer's guide and some examples are available online [158].

### 4.3.1 Modeling DL Program as Graph

In our graph-based approach, a DL program is modeled by a graph instance conformed to the type graph, i.e., meta-model. To fulfill this primary step, we implement the graph generation relying on static code analysis that examines the source code and extracts the valuable code units and segments the information needed to instantiate the type graph's components. This analysis is performed without executing programs to extract the structure

of the DL model from the code. Based on the OMG (Object Management Group) taxonomy of software analysis types [159], our graph generation process belongs to the category of technology-level software analysis because the code inspection routines are further customized to consider the interactions between identified units and connect them to the internals of the used technology (DL library). This provides a more holistic and semantic view of the analyzed DL program, that allows detecting faults related to either DL library's API misuse or DL algorithm implementation requirements. Hence, a specific graph generator should be implemented for each supported DL library. Without loss of its generality, *NeuraLint* currently supports DL programs written using TensorFlow and Keras as two well-known and popular libraries. It should be noted that *NeuraLint* can be extended to detect bugs/issues in DL programs developed by other DL libraries (like PyTorch), as well. The only necessary step is extending the parser to cover specific APIs of each DL framework. Moreover, by employing static analysis we are limited to the information available prior to the runtime, in contrast to dynamic analysis performed on programs while they are executing. Hence, we cannot detect bugs/issues depending on information introduced in the runtime environment, like dynamic types or dynamic constructions. However, we believe that the current approach can detect a significant amount of bugs/issues since DL programming is usually simple, and much useful information on DL models can be extracted by static analysis. In the following, we describe steps of modeling of DL programs as attributed graphs.

Our approach consists in parsing the DL routines called in the DL program line by line to extract the components related to both the DNN model and the DL training algorithm, as well as their configurations. The identified components are independent of the context of the parsed program. We used Abstract Syntax Tree (AST) to parse the DNN program script. AST represents the abstract synthetic structure of the scripts as a tree. This tree represents the abstract syntactic structure of the source code of the DL program. Each node of the tree denotes a construct or statement in the source code of a DL program. Arguments of function calls or assignment statements are extracted and stored in subtrees of the node. As we process the code line by line according to AST, the graph is constructed gradually by appending nodes and edges. In each DL library, there are specific built-in routines for defining various layers, configuring them (e.g., adding dropout and activations), connecting layers to each other, feeding input, calculating output, and training the network. In this way, the most important parts in the DL program for constructing its model like layers, dimensions, loss and optimizer functions could be identified by the parser. Algorithm 2 illustrates this process. Based on the information extracted by AST, for any line in the code of a DL program that indicates an assignment statement, we build a dictionary to store its value for further usage. For API calls that add or configure a layer, we firstly extract the related properties like type and

number of units (neurons) from AST or dictionary of variables. Since the current version of *NeuraLint* is designed to handle DL programs developed using TensorFlow or Keras libraries, we have covered the API of these frameworks. Sometimes, a computation phase is required for each layer to process its attributes and attach it to other layers in the graph correctly. For instance, the dynamic shapes of the layer's tensors (i.e., input and output data layer) should be computed. In such cases, the required properties are computed before adding the node to the graph. If an API call relates to compiling or training the model (building the DNN, adding loss and optimizer functions), the optimizer and loss nodes are added to the graph. At the end, we will check all the interconnected layers to verify the coherence of the datashapes flowing throughout the DNN's computational layers and apply the required corrections. Afterward, the generated graph, including all relevant components (nodes/edges and their properties) based on the extracted and computed information is returned.

### 4.3.2 Model-based Verification using Graph Transformations

The verification rules are implemented as graph transformations to process and verify the graph. Each graph transformation applies to the graph if conditions of the rule are violated. Once the DL source code is modeled as a graph, the violations of rules can be detected with a graph transformation tool that executes the sequence of rules over the model of the DL program. In this chapter, we have used the GROOVE toolset [160] to perform graph operations. GROOVE is a tool for implementing, simulating, and analysis of graph transformation systems. It is capable of exploring recursively and collecting all possible rule applications over a host (start) graph. This is referred to as the exploration of the state space of a graph grammar. GROOVE explores the state space by applying a slightly modified version of standard graph traversal algorithms, like depth-first search (DFS) or breadth-first search (BFS). Furthermore, it has a graphical interface for editing graphs and rules, and for exploring and visualising the GTS which could be called via command line, as well. The output of GROOVE is called the final graph on which no further rule application is possible. For more information about GROOVE's internal mechanism and its capabilities for modeling and simulating GTS, the interested reader may refer to [161].

In order to find which rules are violated, the graph transformation system must be simulated. The simulation performed by GROOVE automatically applies the matching transformation rules over the graph of the DL program. Actually, this process generates a state space, in which the model of the DL source code (*graph*) is the start state and the transitions are the applied transformation rules. It explores the state space of all graphs that are reachable from *graph*. In certain states, no more transformation rules can be applied; these states are called final states. A path starting from the start state and leading to a final state, consists

**Algorithm 1:** *NeuraLint*

**Input:** A DL program, *program*, and *rules* as a graph grammar
**Output:** List of bugs or warnings to improve the program
*graph* ← extractGraphFromProgram(*program*) (Algorithm 2)
*final* ← graphChecker(*graph*, *rules*) :

1. starting by *graph*, apply enabled rules.
2. apply enabled rules recursively.
3. terminate when further application of rules becomes impossible.
4. **return** *final*.

*report* ← extractReportFromGraph(*final*)
**return** *report*

of applied transformations indicating the detected faults or violation of verification rules. Moreover, this path indicates the type and location of detected faults. A specific code for each type of fault has been associated with the faulty component when the rule has rewritten the graph.

The rules are implemented in such a way that starts from the first layer and proceeds to the next layers one by one. At first, the general structure and connectivity of deep neural networks is tested assuring that input, hidden and output layers are well-formed and connected. These transformation rules mark the graph components (nodes and edges) with relevant flags to indicate the performed tests. Then, each graph operation checks specific conditions that are asserted in its rule using the information provided in the graph. A transformation should be fired if a rule violation is observed in the model of a DL program. If there are multiple rule violations or various instances of a violation in the considered model, all of them will be detected by applying multiple enabled rules. At last, a parser is developed to process the final graph and extract information about detected issues/bugs to generate a report for the user.

### 4.4 Evaluation

In this section, we report an empirical evaluation that aimed to assess the effectiveness of *NeuraLint*.

**Algorithm 2:** Extracting graph from DL program

```
Input: A DL program in Python developed by TensorFlow or Keras
Output: A graph indicating program's DL model with respect to the meta-model
graph ← empty
dictionary ← empty
for each line of DL program do
    if line encodes an assignment statement then
        extract left-hand (variable) and right-hand side (value) of the statement
        add the statement to the dictionary
    end
    if line encodes a layer then
        extract properties of layer (type, size, kernel, padding, ...)
        look up values of variables in dictionary
        add the corresponding node(s) and edge(s) to the graph
    end
    if line encodes compilation of the model then
        extract properties of compilation (loss and optimizer function)
        look up values of variables in dictionary
        add the corresponding node(s) and edge(s) to the graph
    end
end
return graph
```

### 4.4.1 Studied Programs

We have evaluated the effectiveness and efficiency of *NeuraLint* in detecting bugs/issues on a set of synthetic and real-world faulty DL programs. To create realistic synthetic examples, we also need some real-world DL programs to imitate the faults occurring in them. To find a proper set of real-world faulty DL programs, we have used two main sources: 1) samples found by directly searching over SO with keywords related to the categories of bugs covered by *NeuraLint*, and 2) public datasets of buggy DL programs (from SO and GitHub) released by previous research studies. For the former, we chose SO because it is the most popular Q&A forum for software development. As of May 2020, it has collected more than 19 million questions and 29 million answers. It has been also leveraged by previous studies on DL software systems [11, 12, 138]. Since *NeuraLint* currently supports both TensorFlow and Keras, we searched SO posts tagged by one of these libraries with the objective of collecting buggy DL code including multi-granularity levels such as a single function call, a snippet (few lines) of code or a whole DL program. In fact, we found that SO assembles 57,104 and 27,008 questions, tagged respectively with TensorFlow and Keras, that comprise diverse issues encountered by DL practitioners when dealing with these libraries. Hence, we refined

our search queries with keywords related to the categories of bugs covered by *NeuraLint* that are described in Section 4.2.2 resulting in 255 posts. We manually inspected, for each type of bug, the top-10 relevant SO posts (i.e., according to built-in SO relevance criterion) mentioning one or more of its associated keywords. We consider SO posts, containing full code script or code snippets that are related to one or multiple bugs belonging to the above-mentioned categories. This process left us with 18 faulty DL programs.

Regarding public datasets of buggy DL programs (from SO and GitHub), we consider three publicly available datasets/replication packages [1,11,138]. Also we consider another dataset from a recent research on bug fix patterns in DL programs developed by five popular DL libraries including Tensorflow and Keras [162]. They have studied several repair patterns in DL programs. All these studies investigated various faulty DL programs from SO and GitHub. We have manually inspected all artifacts they have used in their study to find relevant faulty examples to evaluate *NeuraLint*. Actually, finding proper samples for evaluating our tool is not an easy task. We explain the methodology followed and encountered difficulties in the rest of this section. Some DL programs were developed by libraries other than Tensorflow/Keras which are out of scope of the current version of *NeuraLint*. In total, we had 733 SO posts and 682 samples from GitHub from all these sources where 622 programs were developed by TensorFlow and 793 by Keras. Among DL programs developed by Tensorflow/Keras, we have excluded programs containing types of faults that are not covered by the current version of *NeuraLint*, for example those related to recurrent neural networks. So, we were left with 566 faulty DL programs. In the next round, programs developed with older versions of Tensorflow/Keras were discarded if the API related to the fault was not supported in later versions. Many of the remaining samples (89 from SO and 126 from GitHub), especially those from GitHub, were actually libraries (not DL programs) that have been developed on the top of DL libraries for particular problems or domains, e.g. image/speech processing, reinforcement learning, or natural language processing. We also discarded these libraries which were 86. Although their implementation contained bugs that led to buggy DL models, when the libraries were used to build DL models, we discarded them because they do not build a model explicitly using Tensorflow/Keras APIs. For example, they get a specific configuration file or a code written by their own high-level APIs as input, and use it to construct a DL model. Therefore, it is impossible to use those examples since the scope of *NeuraLint* is defined to cover DL programs developed directly by employing Tensorflow/Keras built-in APIs. It should be noted that customized parsers can be developed to extract DL models from any configuration file or high-level code that is not currently covered by *NeuraLint* and then use our tool to find bugs/issues in them. After processing all these artifacts, we ended up with 26 buggy DL programs shared on SO (18 from our direct search and 8 from public

datasets) and 8 from GitHub.

Since *NeuraLint* requires a full DL program to construct a graph on which the model verification is performed, we decided to prepare synthetic examples by a mix of synthetic code and reproduced real DL programs for the evaluation of *NeuraLint*. The reproduction of buggy DL programs from the SO posts is quite difficult when a major part of the code is not provided in the post. Anyway, to reproduce real buggy DL programs, we proceed as follows: (1) we first implement two well-known CNN applications, LeNet [163] on MNIST data [164] and VGG-16 [151] on Imagenet data [143], as base programs. To enhance diversity at technology level as well, we use both of our supported DL libraries, Tensorflow for LeNet and Keras for VGG-16 following, respectively, the official implementations [165] and [166] published on GitHub; (2) regarding implementation-related bugs, we inject each fault found, to one of the base DL programs; (3) regarding design-related issues, we poorly re-designed the structure of the base program's model to include inefficiencies violating the common patterns and best practices mentioned in Section 4.2.2. Finally, we constructed a total of 28 buggy synthetic programs which corresponds to one or two examples per detection rule. We constructed two faulty examples for a rule when there are two contexts in which the rule can be triggered, one example for each of these contexts. For instance, Rule 10, which validates the loss linkage, has been evaluated against both contexts of binary cross-entropy (used for binary class problem) and categorical cross-entropy (used for multi-class problem). For injecting bugs, we followed fault patterns observed in real buggy samples during our rule extraction process (illustrated on Figure 4.3). Hence, the injected bugs are realistic reproductions of faults. Our goal for evaluating *NeuraLint* using synthetic examples is debugging, i.e., making sure of its accuracy and effectiveness prior to evaluating it on real-world examples. For more details, please see our replication package containing all samples and implemented synthetic code [158].

Based on DL bug symptoms defined in [11], we found three bug symptoms in our studied DL programs: *i) Bad performance.* Bad or low performance is a common effect of conceptual issues related to design structure inefficiency or poor choices, misconfiguration of DL components; *ii) Incorrect Functionality.* This symptom refers to situations where the DL program behaves in an unexpected way without any runtime or compilation error. For instance, the DNN outputs only one label among class labels; *iii) Program Crash:* This bug effect is common for all software programs, it means that the program stopped running and raised an exception. Regarding the recommended fixes, we examined the accepted or endorsed answers of SO users to determine the bug-fixing repair (i.e., accepted) or recommendations provided to guide the user who asked the question towards finding the root cause of the error (i.e., open question).

### 4.4.2 Results

First, we have evaluated *NeuraLint* using 28 synthesized examples to investigate the correctness and preliminary effectiveness of the proposed approach. *NeuraLint* has successfully detected the bugs and issues in all synthetic examples. It should be noted that *NeuraLint* extracts the DL model from a DL program and employs graph transformations to apply the proposed rules on the model, not the code, to detect possible bugs/issues. For extracting the model from the code, *NeuraLint* relies on TensorFlow/Keras APIs and not on any particular patterns in the code. Moreover, we have not limited our experiments to these synthetic samples and have tested *NeuraLint* on real-world faulty DL programs. In other words, the tool is evaluated on faulty DL programs with bug patterns that were not considered when creating the tool or synthetic examples.

To evaluate practical effectiveness and accuracy of *NeuraLint*, 34 real-world DL programs from SO posts and GitHub repositories are used. Results are presented in Table 4.1 and Table 4.2. Table 4.1 reports results over DL programs extracted from SO posts. For each DL program, we report the ID of the post over SO, reported symptoms of buggy programs by the developer, fixes recommended by other users, output of *NeuraLint* (violated rules), number of true positive cases and false negative cases respectively. True positives are reported as a+b where a is the number of bugs/issues reported by SO users that are detected by *NeuraLint*, and b is the number of bugs/issues detected by *NeuraLint* that are not mentioned by SO users. For b, two of the authors independently have checked each program and the output of *NeuraLint* manually to ensure that the output is correct. The total number of false positive cases is zero, so we do not report them. It is well-known that the best practice is analyzing SO posts with accepted answers (No. 1 to 20 in Table 4.1) ensuring the proposed solution is a real fix and addresses the mentioned problem. In our searching process for faulty samples, however, we have encountered 6 posts in SO without accepted answers (No. 21 to 26 in Table 4.1) containing relevant DL buggy programs or code snippets. Although none of the provided answers in these posts were accepted by the user who asked the question, we found at least one helpful and correct answer in the posts after a careful analysis. Specifically, one of the authors has manually inspected answers to make sure that SO users pointed out a right solution to the problem according to our verification rules. This process has been verified by another author assuring that we have a correct assessment and that the output of *NeuraLint* is accurate. Regarding samples from GitHub, the results are reported in Table 4.2. True positives are again reported as a+b where a is the number of bugs/issues that are successfully detected by *NeuraLint* according to reported problems in GitHub or a previous research study as mentioned in Subsection 4.4.1. On the other hand, b is the number of bugs/issues detected by *NeuraLint* but not reported in GitHub or a previous study. Similar

to what we have done for SO posts, we have checked each program and the output of *NeuraLint* manually to ensure that the output is correct. In all tables, the rules which detect the bug/issue as reported by the developer are highlighted in bold letters. Detailed information of each sample including the link to GitHub repositories are available in our replication package [158].

In total, 31 out of 44 bugs/issues are detected correctly by *NeuraLint*, so the recall is evaluated as 70.5 %. All of these bugs/issues were identified by users/developers or previous research studies. The recall for SO posts with accepted answers is 76.9 % (20 out of 26), for SO posts without accepted answers is 75 % (6 out of 8), and for GitHub samples is 50 % (5 out of 10). The precision is 100 % meaning that we do not observe any false positive case in our evaluation. Moreover, *NeuraLint* correctly detected 33 additional bugs/issues that were not reported by users/developers who commented on the SO posts or GitHub repositories. Most of them, 29 out of 33, are design issues. *NeuraLint* has successfully detected 64 bugs/issues in 34 real DL programs in overall.

While fewer bugs are detected by *NeuraLint* in GitHub samples compared to SO posts, more design inefficiencies are detected by *NeuraLint* in GitHub samples (14 in 8 samples). Also, since *NeuraLint* is based on a static analysis, being able to detect half of the faults contained in the studied Github projects is already an interesting feat, since it allows developers to catch them early on, before they have to run their programs. Based on these results, we can report that the performance of *NeuraLint* is noteworthy; about three-quarters of known bugs are successfully detected (recall) and a significant number of hidden bugs and design inefficiencies of DL programs. Moreover, its precision is 100 % meaning that while the tool may miss some faults in the evaluated DL programs (overall recall is 70.5 %), it never detects bugs/issues wrongly. The reason is that our detection process is based on proposed verification rules and we report their violations in DL models extracted from DL programs.

We have performed the experiments using a machine with Intel i7-9750H CPU and 16GB of main memory running Windows 10. The average execution time of *NeuraLint* for the studied TensorFlow and Keras samples are 1.800 and 2.049 seconds, respectively. It should be noted that graph checking (performed by GROOVE) consumes the main portion of the execution time, about 99.7 %. Our preliminary analysis revealed that the running time mainly depends on the number of layers of the DL model. Details of the execution time of *NeuraLint* for five real DL programs with different sizes are reported in Table 4.3. According to these results, the execution time of extracting and checking the graph increases as the number of layers grows. Extracting the graph is accomplished by single or multiple passes through the code. Hence, the execution time grows linearly by the number of layers as adding/configuring each layer needs a few API calls in TensorFlow/Keras. On the other hand, GROOVE supports

Table 4.1 Results of validating *NeuraLint* using real DL programs selected from StackOverflow.

| No. | SO # | Symptom | Recommended Fix | Violated Rules | TP | FN |
|---|---|---|---|---|---|---|
| 1 | 44399299 | Program Crash | Change the shape of the input layer | **7** | 1+0 | 0 |
| 2 | 43464835 | Program Crash | Change the shape of the input layer | - | 0+0 | 1 |
| 3 | 42913869 | Program Crash | Change the number of units for the output layer | 3 | 0+1 | 1 |
| 4 | 48518434 | Program Crash | Reduce spatial size of both Conv. filtering and pooling widows | **7** | 1+0 | 0 |
| 5 | 40857445 | Program Crash | Adding a flatten layer | **6** | 1+0 | 0 |
| 6 | 50555434 | Bad Performance | Use softmax activation instead of sigmoid and categorical_crossentropy loss instead MAE | **10** | 1+0 | 0 |
| 7 | 46177505 | Program Crash | Change spatial size of Conv. filtering and pooling widows | 5, 10 | 0+2 | 1 |
| 8 | 50426349 | Program Crash | Change the shape of the input layer | 19, 20 | 0+2 | 1 |
| 9 | 38584268 | Program Crash | Adding a flatten layer | **6**, 21 | 1+1 | 0 |
| 10 | 45120429 | Program crash | Change the number of units for the output layer, Adding a flatten layer | **6, 19**, 10 | 2+1 | 0 |
| 11 | 45378493 | Incorrect Functionality | Use a sigmoid for last layer activation | **5, 10**, 16, 19, 20 | 2+3 | 0 |
| 12 | 45711636 | Program Crash | Use channels_last format for input data | **7** | 1+0 | 0 |
| 13 | 34311586 | Bad Performance | Remove the last layer activation | **5, 10**, 19 | 2+1 | 0 |
| 14 | 50079585_1 | Bad Performance | Use softmax activation instead of sigmoid and categorical_crossentropy loss instead binary_crossentropy | - | 0+0 | 1 |
| 15 | 50079585_2 | Incorrect Functionality | Change the number of units for the output layer | **10** | 1+0 | 1 |
| 16 | 51749207 | Bad Performance | Use of sigmoid activation instead of softmax | **5, 10**, 19 | 2+1 | 0 |
| 17 | 53119432 | Program Crash | Adding a flatten layer | **6**, 19 | 1+1 | 0 |
| 18 | 55731589 | Program Crash | Use of 'same' instead of 'valid' for layer padding type | **7** | 1+0 | 0 |
| 19 | 58844149 | Bad Performance | Use of sigmoid as last layer activation | **5, 10**, 21 | 2+1 | 0 |
| 20 | 61030068 | Program Crash | Adding a flatten layer | **6** | 1+0 | 0 |
| 21 | 33969059 | Bad Performance | Change the number of units for the output layer | **10** | 1+0 | 0 |
| 22 | 44184091 | Program Crash | Fix the limit size for input sequence data | 15 | 0+1 | 1 |
| 23 | 44322611 | Bad Performance | Prune the DNN, use RMSprop instead SGD | 10, 20, 21 | 0+2 | 1 |
| 24 | 49117607 | Program Crash | Reduce spatial size of both Conv. filtering and pooling widows | 16 | 0+1 | 0 |
| 25 | 55776436 | Bad Performance | Try Data augmentation, Regularization, filtering spatial size reduction, and DNN Depth Increase | **7, 16, 17, 20** | 4+0 | 0 |
| 26 | 60566498 | Bad Performance | Try Data augmentation and Hyperparameters Tuning | **15**, 16 | 1+1 | 0 |

priority-based rule application as well as various search strategies to explore the full state space, i.e., checking and applying all applicable rules in each state [161]. We have used BFS and priority-based rule application to improve the efficiency. However, the execution time of graph checking grows faster than extracting the graph as the number of layers of the DL model increases. The running time of *NeuraLint* can be improved which is left for future work.

### 4.4.3 Discussion

GitHub samples were developed by advanced developers and are more complex than SO posts. While the scope of *NeuraLint* is defined to cover convolutional architecture as a particular type of FNN designed mainly for classification of 2D images, audio spectrograms, or 3D videos, some of studied GitHub samples have used convolutional architecture for data generation (e.g., extraction of structural lines of images[2]) or text classification[3]. In another sample[4], developers concatenated outputs of multiple convolutional architecture, each layer taking all preceding feature-maps as input, which is not frequent in popular CNNs. Using popular convolutional architectures such as VGG, ResNet, or MobileNet as a part of a DL model or modifying them for particular tasks is also observed in studied samples from GitHub[5]. Although the proposed meta-model is capable of representing these models as FNNs, particular rules must be proposed to find faults and improve the accuracy of *NeuraLint* on these samples. The developer added a dropout layer after dense layers to fix the problem in one of GitHub samples[6]. Although, as mentioned in Subsection 2.2, regularization methods are required to improve the convergence and generalizability of DL models, we need more investigations for proposing a rule to detect lack of enough or proper regularizations.
The focus of the design and implementation of *NeuraLint*, is on faults that relate to structural (architectural) properties of DL programs rather than their dynamic properties that need the programs to be executed. In other words, there are some frequent types of bugs/issues in DL programs that could be detected without dynamic analysis of the DL program [1]. However, the lack of dynamic analysis of DL programs is a limitation of our approach. Such analysis would allow for the detection of runtime bugs and bugs/issues in training/inference of DL models. For example, in program No. 22 in Table 4.1, the mismatch shapes is caused by the size of loaded input size during the execution, it is a runtime bug and could not be detected by the current version of *NeuraLint*. Some other bugs in DL programs need in-depth runtime

[2] https://github.com/hepesu/LineDistiller [3] https://github.com/bwallace/rationale-CNN [4] https://github.com/cmasch/densenet [5] https://github.com/mateusz93/Car-recognition/commit/94b36ea [6] https://github.com/dishen12/keras_frcnn/commit/38413c6

Table 4.2 Results of validating *NeuraLint* using real DL programs selected from GitHub.

| No. | Symptom | Recommended Fix | Violated Rules | TP | FN |
|---|---|---|---|---|---|
| 1 | Bad Performance [167] | Changing the last layer activation | 19, 20, 21 | 0+3 | 1 |
| 2 | Bad Performance [168] | Changing layer dimensions | 19, 20, 21 | 0+3 | 1 |
| 3 | Bad Performance [169] | Changing layer dimensions (padding) | **4**, 19, 21 | 1+1 | 1 |
| 4 | Bad Performance [170] | Adding a pooling layer | **5**, 16, 20, 21 | 1+3 | 1 |
| 5 | Bad Performance [171] | Changing layer dimensions | **19**, 16, 20, 21 | 1+3 | 0 |
| 6 | Bad Performance [172] | Adding ReLU activation to the last layer | **3** | 1+0 | 0 |
| 7 | Bad Performance [173] | Adding ReLU activation to the last layer | **3** | 1+0 | 0 |
| 8 | Program Crash [174] | Changing layer dimensions | 19 | 0+1 | 1 |

analysis. For example, using dropout before batchnorm makes the behavior of DNN different during training and evaluation phases. This is the case for program No. 15 in Table 4.1. In another sample[7], batch size has been modified to improve the performance of the learning phase which cannot be investigated without analyzing the learning performance during runtime. Detecting these faults is currently out of scope of *NeuraLint* and to cover them, DL programs must be experimentally tested to assess their performance for the underlying problem and then detect it.

Another challenge that we faced is related to the multiple releases of TensorFlow library that significantly changed the API functions; which makes the graph generator mainly designed and implemented in regards to the 1.15 version, incapable of detecting some of the required components for versions other than 1.15.

Lack or limited access to real DL programs annotated with possible bugs, design inefficiencies and recommended fixes to evaluate DL testing approaches accurately and effectively could be regarded as a barrier in this line of research. Finally, the current version of *NeuraLint* could find problems in FNNs, particularly CNNs. Other neural network architectures, like recurrent neural networks are out of the scope of this version. *NeuraLint* could be applied to detect bugs/issues in such neural networks by extending the meta-model to capture their properties, i.e., possibility of connections between the nodes that form a cycle. Also, new rules could be defined to detect specific problems in each architecture according to frequent observed bugs/issues or best practices.

[7] `https://github.com/taashi-s/UNet_Keras/commit/b1b6d93`

Table 4.3 Execution time of *NeuraLint* for five real DL programs with different sizes (times are in seconds).

| No. | Number of layers | Running time of Graph extraction | Running time of Graph checking |
|---|---|---|---|
| 1 | 6 | 0.003 | 1.757 |
| 2 | 8 | 0.003 | 1.787 |
| 3 | 12 | 0.002 | 1.836 |
| 4 | 13 | 0.003 | 1.906 |
| 5 | 38 | 0.004 | 3.111 |

## 4.5 Chapter Summary

In this chapter, we have introduced *NeuraLint*, a model-based verification approach for DL programs. As a model-driven approach, a meta-model has been proposed for DL programs at first. We have defined a set of verification rules for DL programs based on the meta-model. A model of each DL program is configured by parsing its code to extract relevant information. Afterward, a graph checking process is performed to verify the model and detect potential bugs or design inefficiencies. Graph transformation systems are used to implement the verification rules and modeling approach. The meta-model is represented by a type graph, DL programs are modeled as host graphs, and graph transformations execute the verification rules. *NeuraLint* has been evaluated using synthetic and real DL programs. The results show that *NeuraLint* effectively detects faults and design issues in both synthetic and real-world DL programs with a recall of 70.5 % and a precision of 100 %. Refining (or even redesigning) the meta-model and graph transformations are required to improve the accuracy and detect false negative cases.

# CHAPTER 5 THE DEEPCHECKER: A DYNAMIC PROPERTY-BASED DL PROGRAM DEBUGGER

Recent research works [11] [1] [138] on DL program bugs show that the faults, leading to program crashes, represent only a fraction of real bugs found in DNN training programs. The vast majority of bugs are due to hidden logic errors and configuration inconsistencies, leading to silent failures that are difficult to detect (since they do not prevent the program from running and producing a model). Indeed, a DNN is trained using the back-propagation algorithm that relies on a loss function to estimate the distance between actual predictions and the ground truth, and then, the estimated error is back propagated through the DNN's learnable parameters to adjust their values in the opposite direction of the loss gradient. In practice, components of the training algorithm are provided as ready-to-use configurable routines by DL libraries, however, reusing these routines to implement a training program for a designed DNN is not straightforward and it can be error-prone. From a fundamental point of view, the backpropagation algorithm can be considered as a leaky abstraction since the details of its implementation are not trivial. To illustrate this point let's consider the basic rule of weights initialization which states that values should be small random numbers. Setting weights' values is not simple and straightforward because the use of dummy random initialization could prevent the DNN from training. In reality, depending on developers' design choices, there is a set of custom weight initializations that have been formally proven to be optimal choices, and hence should be adopted by the developers. Actually, the established debugging practices for the training programs of DNNs reposes on diagnose-via-visualization, which is not sufficient to systematically ensure that a training program is bug-free, and consequently that the trained DNN can be trusted. In fact, researchers [175] have proposed advanced visual analytics systems to complement the loss- and accuracy evolution curves, and help DL developers debug and refine the design of their DNNs. Oftentimes, this requires monitoring individual computational units of the model during the training, such as activation maps or error gradients produced at each layer, and after training, an instance-based analysis is performed to identify misclassified instances from a handful of chosen data instances. However, such visualization based diagnosis techniques require significant human intervention and good expertise on deep learning concepts. Given the internal complexity of a DNN, it is always challenging to select a handful of drawing representations (i.e., suitable for screen display) that should be watched and analyzed, interactively. Additionally, these visualizations can hardly be used automatically to track for regressions after a program update. We believe that full-fledged software debuggers are needed to support DL developers.

In this chapter, we propose *TheDeepChecker*, the first end-to-end automated debugging approach for DNN training programs. To develop *TheDeepChecker*, we gathered a catalog of fundamental algorithmic and development issues in relation with the DNN training programs. Then, we inferred the properties that are violated by these identified training program issues, in order to develop the verification routines that can be used to detect their occurrences during the training process. Next, we developed a property-based testing method that activates the derived verification routines, in order to drive an end-to-end automated debugging process for DNN training programs. To assess the effectiveness of *TheDeepChecker*, we implemented it as a TensorFlow-based testing framework that enables the automatic detection of the identified issues in DNN training programs developed with Tensorflow (TF) library. We rely on the taxonomy of real DL faults elaborated in [1] and further searching on Stackoverflow, to identify the structure of faults occuring in DNN training programs. Then, we inject these faults in clean DNN training programs to create a set of synthetic buggy programs, each one containing a particular fault, aiming at challenging the *TheDeepChecker* in identifying the faults or steering the users to them by detecting precise fault-indicative symptoms. Moreover, we assess the performance of *TheDeepChecker* on a mixed selection of 20 real-world TF buggy programs [138] splitted equally between snippets of code shared on StackOverflow (SO) and bug-fixing commits from DL projects hosted on GitHub (GH). As a DL debugging baseline, we manage to run all the studied buggy DL programs on Amazon SageMaker Cloud ML service while activating its internal Debugger's built-in rules. Results show that *TheDeepChecker* can successfully support DL practitioners in detecting earlier a wide range of coding bugs and system misconfigurations through reported violations of essential DL program's properties. Additionally, the comparison with SageMaker Debugger (*SMD*) highlights the ascendancy of *TheDeepChecker*'s on-execution validation of DL properties over *SMD*'s rules verification on training logs in terms of detection accuracy and DL bugs' coverage.

**Chapter Overview.** Section 5.1 presents common training program pitfalls discovered by DL researchers and experienced by DL developers. Section 5.2 introduces our proposed property-based debugging approach for DNN training programs alongside their derived verification mechanisms and its TF-based implementation. Section 5.3 reports about the empirical evaluation of our proposed debugging approach and Section 5.4 comments on the results. Section 5.5 discusses the threats to validity. Finally, Section 5.6 summerizes the chapter.

## 5.1 DNN Training Program: Pitfalls

Many issues can prevent the fitting process of a DNN from performing properly, i.e., finding the best-fitted model. In this section, we elaborate on some of the common pitfalls in designing and implementing DNNs, while pointing out and discussing concrete examples of their derived non-crashing bugs.

### 5.1.1 DL Faults Investigation

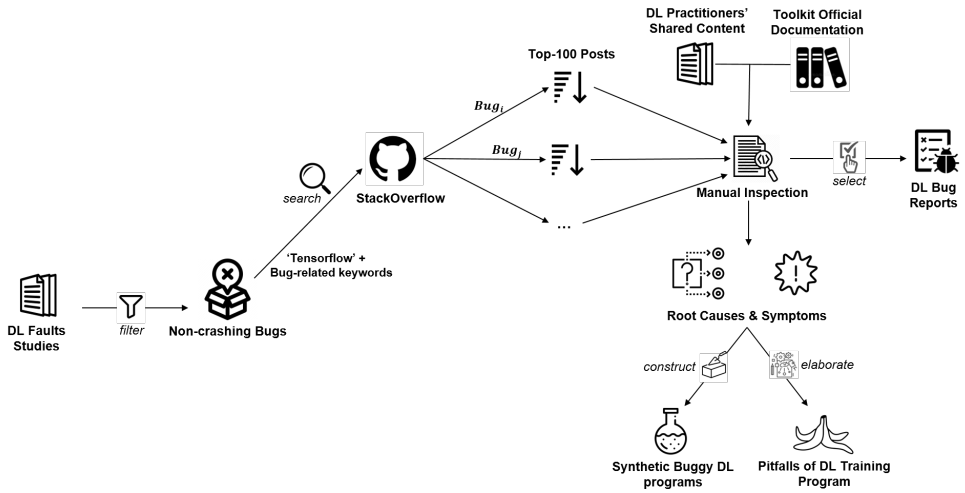



Figure 5.1 Overview of DL Pitfalls Investigation Process

Since deep learning has been increasingly leveraged in diverse real-world applications, researchers have been interested in studying the software development challenges for this next generation of software, including the faults' taxonomy and bugs characteristics. Zhang et al. [138] manually inspected the real-world SO and GH's Tensorflow programs and identified some of the DL bugs, their root causes and symptoms at high level. Then, Islam et al. [11] extended the investigation by including DL programs written with other competitive libraries such as Pytorch and Caffe, and studied furthermore the categories of bugs and their relationships. More recently, Humbatova et al. [1] refined the former bug investigation [11, 138] into a taxonomy of real faults that occur in DL software systems. The taxonomy was deduced from 375 labeled buggy DL code examples built using three popular DL libraries: Tensorflow, Keras and Pytorch. Moreover, the construction of the taxonomy was built in collaboration with 20 DL developers and validated by a different set of 21 DL developers who confirmed the relevance and completeness of the identified categories. Indeed, a bunch of the reported bugs were caused by either coding mistakes, model design issues, or wrong

configurations, that share a common high-level symptom of inefficient training. The latter manifests through convergence difficulties, preventing partially or even totally the training program from fitting the data. Hence, these non-crashing bugs are unique to the deep learning software systems that do not raise exceptions but adversely affect the training dynamics and results. Thus, we aim to identify and understand the development pitfalls and root causes behind the non-crashing DL bugs. First, we started by filtering them from the DL faults collected and reported in the former studies' datasets. Mainly, we discarded the two categories of Tensors&Inputs [1] and GPU usage [1] that represent, respectively, crash-inducing bugs and GPU-related bugs. We focus on detecting the non-crashing bugs among the remaining three categories of Model [1], Training [1] and API [1] that contain, respectively, different misconceptions of the model, multiple poor coding/configuration bugs in the training algorithm implementation, and misuses of the DL libraries' API. Then, we extend the selected subset of bugs with more Q&A posts from Stackoverflow that are related to this family of bugs. We conduct a keyword-based search on StackOverflow (SO) with queries in the form of 'bug_type-related keywords+Tensorflow' and we select, for each bug type/query, the top-100 SO posts (sorted by SO internal relevance criterion). Next, we inspect manually the SO post content including the shared code snippets and users' comments, with the aim of identifying more instances of the studied faults in Tensorflow. Therefore, we found 155 bug reports in relation to occurrences of our targeted DL faults in Tensorflow DNN programs. Overall, only 8% of the bugs in the three above-mentioned categories from [1] are included directly into our dataset, but as we searched using their keywords on SO, 38% of them represent same bug types/root causes in our datasets. In fact, these buggy DL programs could not be included because they are either related to another DL framework (Pytorch, Keras, etc.) or missing necessary information for reproduction such as training examples or hyperparameters' values.

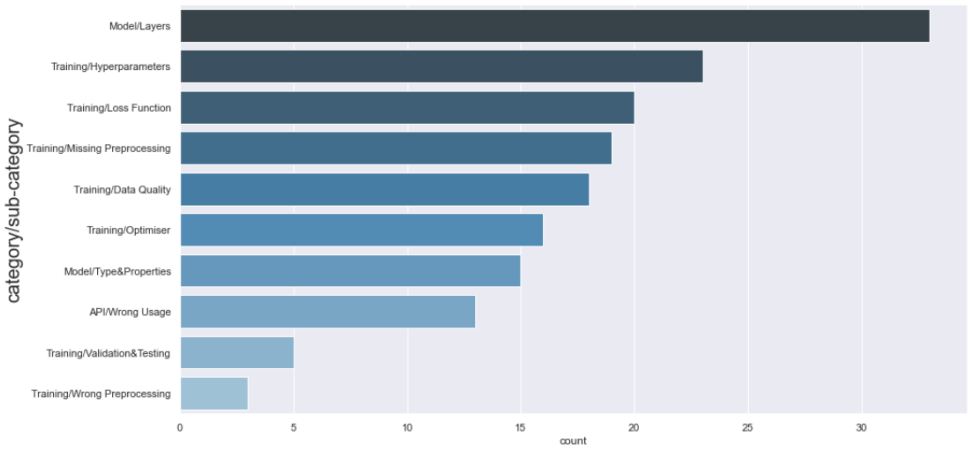



Figure 5.2 Distribution of the collected Tensorflow Bugs over the Taxonomy of DL Faults [1]

Figure 5.2 shows the categories and subcategories of the bugs we collected in our datasets with respect to the taxonomy of DL faults [1]. During the manual inspection of these bugs' instances, we abstract their main root causes and their typical symptoms (i.e., the negative effects observed on the training dynamics and the produced DL model), relying on the practitioners' shared content on troubleshooting and debugging the DL training algorithms including blog posts [176–180] and popular forum discussions [181, 182]. Indeed, the decontextualization of DL bugs from the leveraged API version, the used data and the targeted application, allows recognizing firstly the design or implementation pitfalls that can be the origin of these training issues. Lately, the concrete occurrences of DL pitfalls would serve us in the creation of the synthetic buggy examples (section 5.3.1), which have been used to evaluate the effectiveness of our debugging approach on detecting the targeted DL-specific bugs. Figure 5.1 illustrates the schema of the above-mentioned steps to systematically enrich the datasets of non-crashing bug reports, as well as, identify their main root causes and symptoms. In the following, we present a comprehensive review of the DNN training pitfalls, organized in groups based on the main problematic component of the DNN training program.

### 5.1.2 Input Data-related Issues

A DNN training program implements a data-sensitive algorithm whose inner logic is learned from the training data and generalized to future unseen data. Poor training data quality often translates into an unstable and inefficient training process. Below, we detail the training issues in relation to the input data and DNN components making use of them.

#### Unscaled Data

The scale of DNN inputs and outputs [149] is an important factor that affects the quality of the training. In fact, larger scale input features produce larger intermediate activations, and consequently, larger gradients regarding the weights connected to these over-scaled input features compared to others. Similarly, over-scaled predicted quantities would generate larger errors and gradients. Inversely, an abuse of data re-scaling penalizes the quantities with initially a small range of values. Both situations will induce a pathological loss curvature and an ill-conditioned loss minimization problem. As a result, the risk of gradient unstable phenomenon [183] increases. Modern deep neural networks deploy inner normalization techniques [29, 34] to overcome unstable distributions of the computed activations and gradients, however, their optimization routines should cope with high update oscillations during the early stage of training because of the unscaled data, and as a result, the DNN is firstly trained on how to scale and shift intermediate calculations into an appropriate range. This

overhead complexity slows down the training procedure and might prevent the convergence towards the best-fitted model.

### Distribution-Shifting Augmentation

Given their high learning capacity, DNNs require relatively large and sufficient training data to avoid simply overfitting the data. Since many application domains lack the access to big data and because gathering data is expensive, DL developers often resort to data augmentation techniques [184] to increase the quantity and diversity of their training data. Examples of data augmentation techniques for images include geometric transformations, color space augmentations, kernel filters, random erasing, and cropping. Nonetheless, the use of inappropriate augmentation rules, as shown in the SO posts #57275278, #48845354 and #55786384, can induce a shift in the training data distribution that prevents the DNN from learning effectively. A DNN trained on noisy, shifted data is often hard to converge to a stable state and also incapable of predicting correctly on unseen data (i.e., validation or testing datasets).

### Corrupted Labels

The data used for training supervised machine learning problems are composed of features $X$ (predictor inputs) and labels $y$ (supervised outputs to predict). The DNN's loss minimization problem is non-convex with several possible local minima; so standard gradient descent often falls into those minima because of the unchanged input data $X$ over all the training iterations. To overcome this problem, a mini-batch gradient descent with shuffling has been used to train DNNs, as introduced in 2.1.1. Indeed, shuffling the data instances and performing the gradient estimation on only a subset of them, makes the batch inputs $X_b$ change with every iteration. This helps the optimizer to avoid sub-optimal local minima with relatively noisy and frequent updates to the DNN's parameters. In the implementation, we handle the features $X$ and labels $y$ in separate data structures because the labels should be used only for estimating the loss and performance metrics in supervised learning problems. A common bug in the data shuffler or mini-batch loader, as reported by SO users in #47866803, #46136553 and #41864333, consists of inducing mistakenly a mismatch between features and labels. This represents a particular case of a more general DL issue of corrupted labeled data, which has been extensively studied in the machine learning community (e.g., [185]).

#### Unbalanced Data

Very often in classification problems, there is an unequal number of instances for different labels. An unbalanced dataset biases the predictions towards the majority class or group of labels. Various mitigation techniques have been proposed to address this issue, e.g., over-sampling, under-sampling, or weighted loss function [184]. DL developers should be aware of this situation, whenever it exists, in order to use earlier a mitigation technique for class imbalance or improve the performance measure to capture fairly mispredictions for underrepresented classes. Otherwise, a biased model could be selected based on the overall accuracy among all classes, resulting in erroneous or unfair behavior when dealing with instances that belong to ones that are underrepresented.

### 5.1.3 Connectivity and Custom Operation Issues

To implement a DNN training program, DL developers use DL libraries that allow constructing the computational graph, where nodes and edges represent, respectively, operations and data paths. The operations represent the computational units that form the linear computations, activations, and gradient estimations. Data paths interconnect the operations and allow data to flow from one operation to the next, in order to successfully train and use the model. Through program code, DL developers use library's built-in and newly-implemented components for operations and connect them by either feeding one's outputs as inputs to another or by performing a math operation joining them. Configurable routines enable rapid development and expansion of reliable DNN programs, however they may lead to spaghetti code that becomes too large with scatter variables and glue code (build bindings between components), increasing the risk of coding errors.

#### Network Disconnections

The most basic dependency is between the inputs and the outputs of the DNN. The DNN should predict the outputs based on the information distilled from the inputs. Thus, a DNN training program that does not consider the inputs when performing its internal computations is definitely erroneous. Moreover, disconnections can occur between the intermediate layers. Indeed, DL engineers can forget to connect some branches of the DNN or to pass the right inputs to the layers. When such omissions occur, one or more DNN layers are accidentally removed. A DNN with fewer layers than necessary can still be trained. A DNN can converge to an acceptable performance with only partial layers. If this occurs, however, the program will no longer comply with its specifications and its performance may be severely impacted.

An illustrative example [186] of a connectivity bug occurs when cloning multiple times the code block for constructing a layer, the DL developer may forget to change the input and output for one of these constructed layers which makes it disconnected from the neural network.

#### Incorrect Custom Operation

The common abstractions used by computational units to encode numerical data are tensors, which are multidimensional arrays with supported algebraic operations. These tensors make it easy to manage high dimensional parameters and perform operations on them efficiently. However, the translation of math formulas from scientific pseudo-code to tensor-based operations can be error-prone. As an illustration, let's consider the cross-entropy loss which is a matrix-matrix operation that accepts the probabilities matrix and the matrix of one-hot encoding labels in order to estimate a particular distance. A buggy loss function may not correctly broadcast the operation if the reduction is done over the wrong axis (e.g., sum over rows instead of columns) and mix information between independent data instances of the batch. This introduces an incorrect dependency to the loss function. This issue can be difficult to detect since the DNN can still train and converge poorly and in the best case, can learn to ignore data coming from other batch elements. Besides, DL libraries include an automatic differentiation module that generates the analytical formula and computes the gradient automatically. However, DL developers can include non-differentiable or problematic operations in their custom function as shown in the SO posts #41780344 and #54346263, which negatively affect the gradients flowing over the newly-designed DNN. In similar way, DL developers can also hand-crafted the gradient calculation for their custom operations, but these gradients implemented from scratch should be tested carefully to avoid wrong computations as motivated by the SO posts #46876063 and #64172765.

### 5.1.4 Parameters-related Issues

DNN parameters represent the weights and biases of a DNN's layers. These parameters are randomly initialized, then, they are optimized during the training process. In the following, we discuss pitfalls in the initialization of parameters that can affect their learning dynamics.

#### Poor Weight Initialization

An improper initialization of the weights for a DNN hampers the stability of the learning optimization problem, leading to unstable activation during the forward pass and unstable

loss gradients of the backward flow. First, the constant weights induces a symmetry between hidden neurons of the same layer. Thus, the hidden units of the same layer share the same input and output weights, which makes them compute the same output and receive the same gradient. Hence, each layer's neurons perform the same update and remain identical; i.e., wasting capacity. Second, random sampling of initial weights breaks the symmetry between the neurons, however, the quality of training is strongly affected by the choice of initialization [25]. Indeed, the derivative equations 2.8 and 2.9 show that the estimated gradients include multiplication by weights, which makes their initial magnitude scale affects their growth or decay over iterations and might induce exploding or dead weights [187].

#### Ineffective Bias Initialization

A bias is like the intercept added to a linear equation. Its main purpose is to allow degrees of freedom close to the origin, which improves the representation capacity of a neural network; so it can fit better to the given data. Generally, the initial biases are always set to zeros. Despite this, null bias for particularly-skewed data distributions (e.g., unbalanced datasets) slows down DNN training, which would do the bias calibration during its first few iterations. It means that non-zero bias could contribute significantly to fit the model if it is delicately set up to approximate the bias of data. For example, learning a classification problem with a rare label is a kind of bias already known; so the final layer's bias should be carefully initialized to accelerate the learning task.

### 5.1.5 Activation-related issues

Activation represents the intermediate computation that introduces non-linearity to filter the information computed by the previous layer. In the following, we discuss some problems related to activations.

#### Activations out of Range

Activation functions are nonlinear functions that determine the output of a neural network. The function is attached to each neuron and determines whether it should be activated ("fired") or not, based on the relevance of neuron output for the model's prediction. Activation functions also help normalize the output of each neuron by transforming inputs into outputs that are within a predefined range of values. When DL developers implement an activation function from scratch, there is a risk of bugs that leads to a wrong or unbounded mathematical function yielding outputs within a range inconsistent with what is expected

by the developer (e.g., sigmoid's outputs are between $[0, 1]$ and tanh's outputs are between $[-1, 1]$).

### Inadequate Hidden Layer Activation

Although the choice of hidden activation function is a design engineering problem, it is not an empirical and performance-driven selection because there are activations that are more suitable and even specialized for particular use cases rather than others. In fact, a non-linear activation is an essential design component; so a bottom-line inadequate choice would be keeping identity function for activation as evidenced by SO posts #53138899 and #46181692. Nevertheless, softmax is a special non-linear activation designed specially to transform the logits into probabilities; so mistakenly choosing it, as an activation for hidden layers, would likely hinder the parameters learning and often discourage a smooth flowing of gradient, as shown in the SO post #52575271. Below, we detail training issues in relation with well-known hidden activations that could happen in certain design circumstances, where selecting an alternative activation function should be considered.

**Saturation of Bounded Function** Activation functions with a bounded sigmoidal curve, such as sigmoid or tanh, exhibit smooth linear behavior for inputs within the active range and become very close to either the lower or the upper asymptotes for relatively large positive and negative inputs. The phenomenon of neuron saturation occurs when a neuron returns only values close to the asymptotic limits of the activation functions. In this case, any adjustment of the weights will not affect the output of the activation function. As a result, the training process may stagnate with stable parameters, preventing the training algorithm from refining them. In fact, we can write the equation index-free to illustrate the gradient computation flow in general:

$$lossW = a_{in} \times \delta_{out} \tag{5.1}$$

where $a_{in}$ is the activation of the neuron input to the weight $W$ and $\delta_{out}$ is the error of the neuron output from the weight $W$.
When the activation function $\Phi$ is saturated, its outputs are in the flat region where $\Phi' \approx 0$; so $\delta_{out} \approx 0$ and $W$ freezes or learns slowly.

**Dead ReLU Function** ReLU stands for rectified linear unit, and is currently the most used activation function in deep learning models, especially CNNs. In short, ReLU is linear (identity) for all positive values, and zero for all negative values. Contrary to other bounded

activation functions like sigmoid or tanh, ReLU does not suffer from the saturation problem because the slope does not saturate when $x$ gets large and the problem of vanishing gradient is less observed when using ReLU as activation function. Nevertheless, ReLU risks "dead ReLU" phenomenon [187] because it nullifies equally all the negative values. A ReLU neuron is considered "dead" when it always outputs zero. Such neurons do not have any contribution in identifying patterns in the data nor in class discrimination. Hence, those neurons are useless and if there are many of them, one may end up with completely frozen hidden layers doing nothing. In fact, given the index-free Equation 5.1, we can see that when the activation is zero $a_{in} = 0$, the loss gradient w.r.t weights becomes zero too ($lossW = 0$); therefore $W$ freezes and no longer receives updates. This problem is often caused by a high learning rate or a large negative bias. However, recent ReLU variants such as Leaky ReLU and ELU are recommended as good alternatives when lower learning rates do not prevent this issue.

**Inadequate Output Layer Activation**

Concerning the output layer, the activation function should map the internal calculated results into valid predictions. In case of mismatch between the ranges of last activation layer's outputs and ground truth labels, the model could not learn a correct mapping function since it is not able to produce the full range of possible outcomes.

**Classification Outputs** The model is learning to predict probabilities, so sigmoid and softmax are the best candidates for, respectively, binary and multinomial classification. For instance, a missing softmax layer prior to the cross-entropy calculation can lead to performance degradation and numerical instability issues as evidenced by the SO post #53254870. However, the use of softmax for a classifier model with 1-dim output leads to the incapacity of outputting the full range of class labels, as evidenced by the SO posts #59129802 and #53971451 where sigmoid should be used to output the negative class 0, or #51993989 where tanh should be used to output both of labels: 0 and $-1$. Other common pitfalls are stacking consecutive output activations, which add useless computation levels that may erase relevant learned information, obstruct the natural gradient flow, and adversely affect DNN performance. Indeed, the redundancy activations were often result from: *(1)API misuse*, where recently-provided stable API loss functions with the logits activation, softmax or sigmoid, included (i.e., *tf.nn.softmax_cross_entropy_with_logits* and *tf.sigmoid_cross_entropy_with_logits*), mislead several DL practitioners that passed the result of last layer activation to these loss functions, which resulted in a double application of softmax or sigmoid on the outputs (e.g.,

we refer to SO posts #36078411, #46895949, and #42521400); *(2)misconception of abstractions*, the definition of a function that abstracts the creation layers with some parameters to facilitate stacking the neural network's layers, however, useless non-linear activation can be applied to the last layer before probabilities transformation by mistake, which restricts the range of outputs. For instance, a ReLU activation before applying softmax, as happened in the SO post #44450841, would nullify negative values and make all their corresponding labels share the same probability after applying the softmax.

**Regression Outputs** The last activation should map the internal computations into a range of values that equals (or is the closest) to the actual interval of target outputs, in order to ease the optimization process. For instance, SO posts #60801900 and #64998875 show how the use of Relu, having an output range of $[0, +\infty]$, prevents the estimation of negative targets and the SO user in the post #62313327 should switch from sigmoid (i.e., outputs values within $[0, 1]$) to tanh (i.e., outputs values within $[-1, 1]$) to meet the real range of ground truth labels.

### Unstable Activation Distribution

The activations encode the representation of features detected at that processing layer during the training process. Thus, the fired activations indicate that the DNN already detects low-level features, which could be relevant for following layers. That is why the stagnation of activations caused by saturation or dead phenomenon hinders the capacity of the DNN to learn useful patterns from the data. Similarly, over-activated layers that are active for all inputs and unstable activation layers that have high variability in their values can lead to numerical instability and–or divergence problems. In fact, activations represent the input features of the next layer. The internal computation of this layer adjusts the parameters in order to infer patterns from features (i.e., activations).The internal computation of this layer adjusts the parameters in order to infer patterns from features (i.e., activations). By analogy to the input normalization, the distribution of the intermediary detected features (inside the DNN) is important to ensure an effective optimization using backpropagation of loss gradient through layers' parameters. More formally, the index-free partial derivative formula 5.1 shows well how the magnitude of activations affects directly the magnitude of weight updates. Researchers [29] [34] have proposed different techniques to normalize the outputs of hidden layers and obtain activations with zero mean and unit standard deviation. Concretely, these additional internal scaling transformations are important to control the magnitude of the gradients and improve, formally, the $\beta$-smoothness and the Lipschitzness

of the estimated loss.

### 5.1.6 Optimization-related issues

The optimization of DNN's learnable parameters consists in minimizing, iteratively, the loss, *i.e.,* empirical error of DNN's predictions regarding supervised training data. Actually, gradient-based algorithms such as SGD, Momentum, and Adam, are the preferred way to optimize the DNN's internal parameters. Next, we discuss several issues that impede the optimization process, while describing our proposed verification routines to catch them earlier.

#### Wrong or Inappropriate Performance Measurements

The iterative optimization of parameters often converges to an equilibrium behavior of the DNN. At this point of equilibrium, the optimal or near-to-optimal DNN status is reached. To find this best-fitted DNN (i.e., highest accuracy or lowest absolute error), the DNN training algorithm acts indirectly by minimizing a loss function estimated on the training data with hope of improving the performance of the on-training DNN. Hence, the loss is primarily designed to measure the distance between predictions and real outputs, while it should respect fundamental properties of an objective function for first-order gradient optimization. Empirical loss minimization for DNN training works well when the minimized loss represents the fitness of the DNN relative to the data. Thus, a wrong loss function with regards to true model risk, misleads the training algorithm that, despite its success to reduce the loss, could not improve the target performance measure (e.g., classification accuracy). For instance, inadequate choice of loss function like choosing mean squared error (MSE), which is a standard loss for regression problems, to compute the deviation between predicted probabilities and target class in a classification problem (e.g., SO posts #38319898 and #50641866). Another common fault in relation with the loss is the use if ineffective loss reduction strategy like in these SO posts where there are no reduction at all (#36127436) or a sum instead of mean reduction (#43611745 and #41954308). Indeed, the reduction strategy allows to aggregate the losses computed for all of the data instances into a scalar loss value. The aggregation could be the average or the sum, however, mini-batch gradient descent variants are commonly used for minimizing the DNN's non-convex loss function. Hence, mean reduction is better than sum reduction, because averaging losses over the mini-batch would keep the magnitude of loss independent of the batch size and of other hyperparameters that are also sensitive to the magnitude of loss gradients like the learning rate. Besides, a mistaken performance metric also regresses the expected covariance between both training quality measures (e.g., loss and accuracy). For instance, bad choice of accuracy metric with respect to the problem

would yield illogic performance measurements and it can be the use of classification accuracy rate for a regression problem, the use of multiclass accuracy metric for a binary classifier, or inversely, the use of binary classification accuracy metric for a multinomial classifier, as evidenced by the SO posts respectively, #62566558, #62354952 and #42821125.

**Inadequate Learning rate**

As shown in the update equations 2.10 for weights and biases, the predefined learning rate controls the magnitude of update at each step; so setting the learning rate too high or too low can cause drastic changes to the optimization process and cause several erroneous behaviors. A high learning rate would push the layer's parameters changing rapidly in an unstable way; preventing the model from learning relevant features. The intuition is that the parameters are a part of the estimated mapping function, so we risk overfitting the current processed batch of data when we try to strongly adapt the parameters in order to fit this batch. An excessively-high learning rate may lead to convergent loss minimization and numerical instability by having NaN loss or output values. Inversely, a low learning rate can slow down the parameters changing; making it difficult to learn useful features from data, and consequently, the minimization process may not converge to a steady state and may even experience a non-decreasing loss value during training. Starting by ineffective learning rates is very common when the DL developer is dealing at first time with a learning problem as evidenced by these SO posts #42264716, #62381380, #55718408, #34743847, #47245866, #40156629 and #59106542.

**Unstable Gradient Problem**

The loss gradient equations 2.8 and 2.9 show that the gradient of a layer is simply the product of errors back-propagated from all its next layers (i.e., following the forward direction). Intrinsically, the layers tend to learn at different speeds and deeper neural networks can be subject to unstable situations if no advanced mechanism is applied to balance out the magnitude of gradients. However, the unstable gradient problem could be more severe and could manifest in the form of vanishing or exploding gradients, as described below, due to poor design choices of initializations and hyperparameters.

**Vanishing Gradient** In this case, the gradient tends to have smaller values when it is back-propagated through the hidden layers of the DNN. This causes the gradient to vanish in the earlier layers, and consequently, it would be nullified or transformed to undefined values such as Not-a-Number (NaN) caused by underflow rounding precision during discrete executions

on hardware. The problem of vanishing gradient can lead to the stagnation of the training process and eventually causing a numerical instability. As an illustration, we take the example of a DNN configured to have sigmoid $\sigma$ as activation function and a randomly initialized weight using a Gaussian distribution with a zero mean and a unit standard deviation. The sigmoid function returns a maximum derivative value of $\sigma'(0) = 0.25$ and the absolute value of the weights product is less than 0.25 since they belong to a limited range between $[-1, 1]$. Hence, it is apparent that earlier hidden layers (i.e., closer to the input layer) would have very less gradient resulting from the product of several terms that are less or equal to 0.25. Therefore, earlier layers receiving vanishing gradients would be stagnant with low magnitude of weights' changes.

**Exploding Gradient** The exploding gradient phenomenon can be encountered when, inversely, the gradient with respect to the earlier layers diverges and its values become huge. As a consequence, this could result in the appearance of $-/+\infty$ values. Returning to the previous DNN example, the same DNN can suffer from exploding gradients in case the parameters are large in a way that their products with the derivative of the sigmoid keep them on the higher side until the gradient value explodes and eventually becomes numerically unstable.

Therefore, advanced mechanisms like batch [34], layer [29], weight [188] normalizations and tuning of optimization hyperparameters such as learning rate or momentum coefficients, are needed to provide adaptive gradient steps and to establish relatively similar learning speed for all the neural network's layers.

### 5.1.7 Regularization-related issues

The regularization strategy prevents the model from overfitting the data, while allowing the DNN to acquire enough learning capacity to learn useful patterns and fit the data properly. In the following, we introduce potential issues related to incorrect and ineffective regularization techniques.

#### Lack of-or-incorrect Regularization

Regularization techniques [25], including penalty cost on the weights magnitude and specialized DL regularization like dropout, discourages the optimization from exploring complex models and exploiting spurious correlations in favor of reducing further the loss. Thus, lack of regularization leads to a noiseless training process with high capacity modeling that risks capturing even residual variations in the given sample of data and tends to overfit it quickly.

Concretely, as shown in Equation 2.12, a zero or very low $\lambda$ makes the penalty cost useless with no effect on the objective loss function, which enables the free-growth of weights and intensifies the threat of overfitting even coincidental noises in the sampled batches used for training the parameters. Regarding dropout, a high retainment probability of neurons (*pkeep*) for wide dense layers or large activation maps for convolutional layers decreases the size of the dropped subset of neurons, which eliminates the randomness effect introduced between the input features at each inference calculation. Concerning batchnorm, a common mistake (e.g., SO posts #43234667 and #52279892) consists in forgetting or misusing the routine that ensures the continuous estimation of the moving average $E[x]$ and variance $Var[x]$ during the training. Moreover, batchnorm makes the loss function dependent on the batch size because an instance of the batch can affect the batch mean and variance estimated, and consequently, affect both activations and loss values for other instances in the batch. For example, the use of unit batches, as reported in the SO post #59648509, should not be applied because the batch variance would be zero and relatively small batches would increase randomness and make the statistics estimation noisy. Thus, the DNN fails to perform any normalization on the intermediary calculation results at the inference time, which means that the regularized version of DNN becomes incorrect and its inner calculations are non-representative. In case of mixing dropout and batchnorm, Li et al. [145] reported a disharmony issue, named variance shift, between dropout and batchnorm when applying dropout first. Indeed, the population statistical variance estimated by batchnorm on the entire DNN training becomes inconsistent and non-representative because of the shift variance of weights done by the dropout when the DNN is transferred from the training to the testing mode. In other words, dropout proceeds by randomly removing the information coming from a subset of neurons to prevent possible neurons' co-adaptation. Thus, we have to pass the cleaned information (i.e., after dropping out some neurons) through batchnorm statistics estimator, otherwise, the statistics would be biased by considering all the neurons, i.e., dropped ones included. A common symptom for all the above issues is that validation/testing error rates are higher than training error rates. However, this symptom is quite connected to overfitting situations; so more fine-grained symptoms are necessary to guide the users towards the occured issue.

**Over-Regularization**

Inversely, a too strong regularization (with high $\lambda$) can significantly reduce the magnitude of weights, which may result in underfitting; leading to useless, dead weights, as discussed in the SO posts #51028324 and #51028324. Regarding dropout, a low *pkeep* will be considered as a strong regularization because it introduces too much randomness and may prevent the

DNN from convergence to a stable state; so *pkeep* should be tuned carefully considering the underlying DL problem and the depth of the designed neural network. For instance, many SO posts #60591577, #44832497, #46515248, #44695141, and #64289118 reported poor training convergence and model performance resulting from inappropriate configuration of dropout layers. Indeed, it has been shown that *pkeep* cannot go below the minimum of 0.5, which represents the maximum additive regularization [32]. In practice, the fixed *pkeep* for hidden layers, and especially, layers close to the output layer, is recommended to be within $[0.5, 0.8]$. However, *pkeep* for the input layer should be kept to about 0.8 or higher (i.e., closer to 1.0). In fact, during the test time, to compensate for disabling the dropout, the learned parameters are scaled by the *pkeep* factor. However, Gal et al. [189] has shown that this approximation at test time becomes noisier and less accurate as the underlying layer is far from the output; which can explain the high risk of instability induced by low neurons' retainment probability for the earlier layers, especially, convolutional layers in CNN and the input layer. Therefore, the common effect of strong regularization is having low error rates for training/validation/testing data, however, this indicates an underfitting situation in general. In our debugging methodology, we propose verification routines to target accurately these over-regularization issues.

## 5.2 DNN Training Program : Property-Based Debugging Approach

In the previous section, we have presented DNN training pitfalls related to misconfigurations and coding bugs, organized by problematic components, to comprehend their symptoms and their negative effects on the dynamics of training and the quality of optimization. In this section, we introduce an adaptation of property-based software testing (PBT) that assembles the verification routines that we propose for debugging DNN training programs.

### 5.2.1 Property-Based Model Testing

It has become mainstream research to focus on DL testing approaches that explicitly check for model properties. Regarding security [190] and robustness [191], the learned function should be Lipschitz, such that small perturbations to the input are guaranteed to spawn bounded changes to the output. In a similar way, other important properties have been proposed to satisfy the desired requirements in terms of privacy [190] and fairness [192]. Indeed, the validation of a trained DL model's properties is quite similar to traditional property-based testing (PBT) because it consists of verifying that the implemented/trained deterministic function, mapping the inputs to the outputs, satisfies the desired properties for all the valid inputs. However, the straightforward application of PBT to the DNN training program, which

is a stochastic data-driven optimization algorithm, appears to be quite challenging. In fact, the DNN training consists of a data-driven, iterative process guided by the backpropagation equations that starts from an initial DNN state and evolves the state following a trajectory dictated by the dynamics of the update step equations until converging hopefully to an equilibrium state. In this section, we introduce an adaptation of property-based software testing (PBT) that assembles several properties of training programs that should be satisfied by the initial state (i.e., pre-training conditions), maintained for the intermediate states of the on-training DNN (i.e., proper fitting conditions), and validated for converged DNNs (i.e., post-training conditions). Besides, an adaptation of input shrinking, as detailed in 5.2.6, can be applied to the DNN state to determine at which level and in which component the property is violated.

### 5.2.2 Origins and Types of DNN Training Properties

Over the last decade, we have witnessed a wide adoption of DL technology in various industrial domains that represent the outgrowth of huge efforts and dedications from the DL community into the knowledge transfer. Indeed, applied DL researchers and experts vulgarize DL fundamentals and research advances in sort of principles, techniques and tricks in order to get more practitioners involved into applying DL technology for solving learning task problems. This gave birth to popular applied DL textbook [25], academic lectures [193, 194], articles [135, 136, 142, 195], and industry courses [196, 197], as well as experts' blogs [176–180] and DL practitioners' forum discussions [181, 182] on DNN troubleshooting techniques and strategies. Figure 5.3 summarizes the steps that we follow to construct automated verifi-

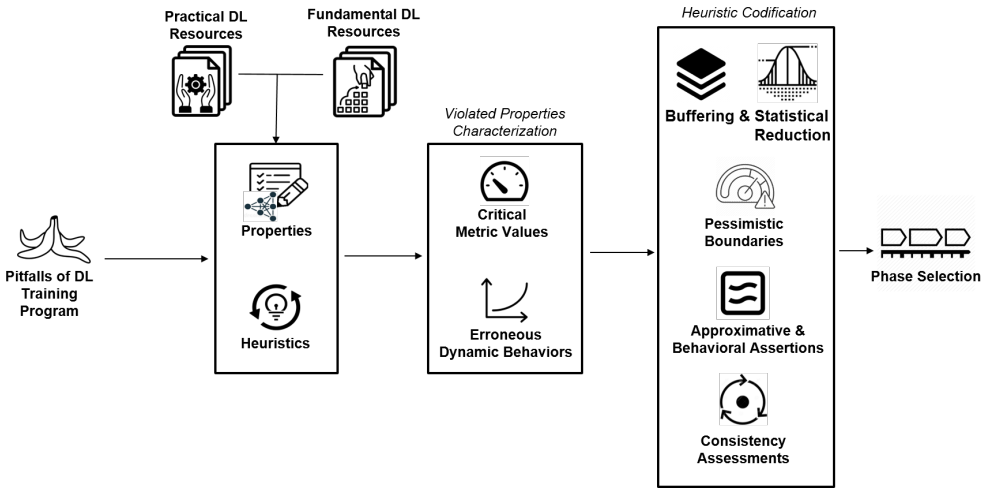



Figure 5.3 Main Steps of Development Process for Verification Routines

cation routines for the identified DL training pitfalls. From the fundamental DL resources, we distill the main properties and design principles for each neural network component for which issues have been detected. For instance, we find the properties in relation to the initial random parameters, the hidden and output activation functions, or the gradient-based optimization routines. Basically, DL training programs are sensitive to the DL bugs that violate some of the involved components' properties. Nevertheless, the detection of properties' violations in a suspicious DL training program is quite challenging. That's why, we explore the heuristics and troubleshooting strategies elaborated by DL experts, which rely mainly on plotting histograms of model internals and curves of performance metrics, in order to spot unexpected distributions and irregular curve shapes. For instance, Glorot and Bengio [24] watched the activation distribution to detect any possible layer saturation when they studied different random weight initializations. Then, we analyze how these heuristics are able to characterize the violations of properties. Generally, the heuristics specify critical values for metrics that shed light on a faulty DL program's state, such as high ratio of null activations or high magnitude updates of parameters. Besides, the heuristics can also describe erroneous training behaviors that would manifest in the dynamics of the metrics, such as unanticipated fluctuating or diverging loss. Even if these abnormal behaviors could be captured and illustrated by experts through visualizations (e.g., histograms of activations, curve of losses over epochs, etc.), the codification of automated verification routines, that detect the violation of statistical learning properties and not-recommended instability for an on-execution DL training program, is challenging. Indeed, the inherent iterative nature and stochasticity of DNN training algorithms makes the regular deterministic test assertions impractical because a single property-violating state is not sufficient for asserting the occurrence of an issue. For instance, the current state may trigger a dead layer (i.e., more than 50% of ReLUs in a layer are null), but the next state following the updates can avoid the problematic situation by reducing the inactive ReLUs. Hence, the persistence of the property-violating state, catched by the heuristics, should be taken into consideration to avoid overwhelming warnings and misleading false alarms during the debugging sessions. Below, we explain the developed guidelines to codify the DL experts' heuristics into robust and dynamic verification routines that are designed to assess persistent behavioral issues.

**Buffering and Statistical reductions**

The parameters and internal computations in a deep neural network are volatile multi-dimensional arrays. Thus, we define buffers to store the last intermediary states, including the hidden activations, the predictions, the losses, etc., in order to validate the heuristics on a set of recent states instead of a single one. Then, we calculate different statistics on

their distributions over different axis of interest to reduce the dimensions and create the most appropriate data views/metrics to handle in the codification of the rule. For instance, the checks on the activation distribution would run periodically on all the activations stored on the buffer, which are obtained from the last training iterations (i.e., anticipated to be using different parameters and batches of data). By default, we set up a buffer size that equals 10. Therefore, given the same example of dead layers, we reduce the buffered activations, from 10 values per neuron to a single one, using $95^{th}$ percentile, which is a robust maximum estimation (highest value under the top 5%). Then, our proposed check would flag the layers with more than a half of dead neurons, which have returned 95% of outputs below the minimum threshold of $1e-5$ for the last 10 training iterations.

**Pessimistic boundaries**

As *TheDeepChecker* consists of a debugging method, we set up pessimistic thresholds to spot critical values and erroneous behaviors that are probably caused by a DL bug. This conservative strategy can effectively reduce many possible false alarms in relation to ineffective training traits that usually manifest in earlier iterations or on complex learning problems. Nevertheless, we keep the thresholds as user-configurable settings in order to adjust the sensitivity of the verification routines on specific DL architectures according to user interests.

**Approximative and Behavioral assertions**

The heuristic-guided DL program diagnostic requires the implementation of approximative assertions including almost numerical equal assertions for floating numbers and statistical significance tests to identify if the obtained program state is outside the anticipated set of possible states by the experts. For instance, there are no best initial parameters, however, DL experts have shown the importance of sampling the random parameters at the first iteration from a carefully-designed distribution, depending on the neural network characteristics. Besides, crafting verification rules for diverse abnormal DNN states such as stagnated loss, huge weights or vanished gradients, etc., can be difficult because the involved metrics' thresholds would vary between models and problems. Thus, it is important to focus on characterizing the erroneous behaviors: stagnation, diverging, or vanishing instead of the resulting faulty state to which the buggy DNN training program would converge. Indeed, we enable the detection of abnormal trends through the assessment of their evolution over consecutive iterations, i.e., by considering a window of steps, generally, window size would be in-between 3 and 5. In the following, we describe our proposed behavioral assertions for the common erroneous behaviors resulting from the identified DL training pitfalls:

**Stagnation** It is the opposite of changing and moving quantity. Hence, we define a minimum percentage difference by which the quantity of interest should change at each step within the window; otherwise, we flag it as stagnated. More specifically, the change direction is already known to compute a relative percentage of increase or decrease, e.g., we expect that the loss keeps decreasing until convergence to a minimum.

**Diverging or vanishing** They represent unexpected huge increases and decreases in a quantity over time. They can be simulated as exponential growth and decay, $q_t = q_0 \times r^t$ where, respectively, $r > 1$ and $r < 1$. Thus, we can approximate the rate of change $r_t = \frac{q_t}{q_t - 1}$ for a window of recent steps, then, we consider that a quantity is diverging if the calculated $r_t$ are higher than a low bound, or it is vanishing if all the $r_t$ are lower than a high bound. In our constructed verifications, we used 2 as low_bound, meaning that the quantity should constantly double its value at minimum to be considered a diverging quantity and inversely, we used $\frac{1}{2}$ as high_bound. The component is flagged when it maintains this unanticipated non-linear evolution for a predefined window of steps.

**Consistency assessments**

Several experts' heuristics, that are incorporated into *TheDeepChecker*'s verification routines, help recognize the unstable distributions of parameters, activations and gradients, as well as unexpected optimization updates and loss curves. However, the riskiness of these unstable learning situations increases when the identified issue persists or becomes more severe over the iterations. Hence, we smooth the verification logic by *(1)* computing the rule's metrics on the aggregation of previously-obtained states from the buffer; *(2)* considering a forbearance period, which is a prefixed number of steps to wait, despite the persistent failure of the verification rule, before flagging the occurrence of the issue.

Finally, we select the debugging phase during which the codified verification routine would run depending on their input DNN states. Indeed, we mainly separate between the initial state verification, the validation of the under-test DL program running on a batch of data, and the need for longer training over larger datasets or comparison of multiple trained models. In the following, we describe the chronological sequence of the debugging phases, as illustrated in Figure 5.4 and we detail the heuristic and logic of all the verification routines included in each phase of *TheDeepChecker*'s debugging session.

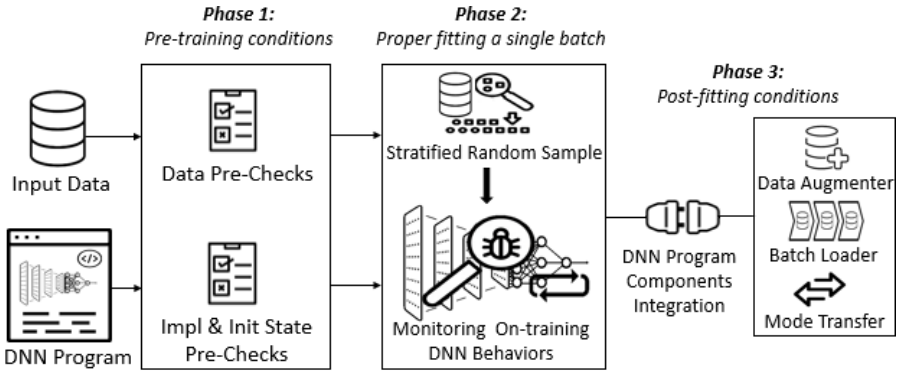



Figure 5.4 Overview of *TheDeepChecker* Debugging Phases

### 5.2.3 Phase 1: Pre-training conditions

The first phase of our debugging process occurs before starting a training session. It enables running static pre-checks of the input data and the starting initial state of the on-training DNN. The benefit of these preliminary verifications is to validate, from the start with a null training cost, the quality of feeding data (i.e., input features, labels), the correctness of essential implemented components (i.e., gradient, custom operations), and the adequacy of the starting state (i.e., initial parameters, first loss). In the following, we describe the pre-checks and their related training pitfalls.

#### Data Distribution

DL practitioners often perform linear re-scaling of the input and output features, in order to adjust their distribution into a common scale without distorting differences between the ranges of original values. In fact, the two most common data scaling techniques are: *(1) standardization* consists in transforming the inputs into z-scores, which means that the transformed data should have a zero mean and a unit standard deviation; and *(2) normalization* consists in re-scaling each input feature using its maximum and minimum elements, to have values within a predefined small range such as $[0, 1]$ or $[-1, 1]$.

**Scaled Data Verification** We extract the data, inputs and outputs, that are fed to the training program; i.e., the final data that have been going through the preprocessing pipeline, and then, verify that the data is scaled properly. In our verification routine, we start by detecting the constant features that have a zero variance. Then, we support checking whether the features are zero-centered with unit standard deviation, or they belong to one of the two well-known normalized range of values: $[0, 1]$ or $[-1, 1]$ [198]. Moreover, this helps detect if

the data accidentally includes undefined or non-finite quantities (i.e., NaNs and Infs).

**Unbalanced Labels Verification** We compute the Shannon equitability index [199], which summarizes the diversity of a population in which each member belongs to a unique group, to estimate the balance between the frequencies of labels. On a data set of $N$ instances, if we have $K$ labels of size $N_k$, we can Shannon equitability index as follows, $\frac{-\sum_{i=1}^{k} \frac{N_k}{n} \log(\frac{N_k}{n})}{\log(K)}$, and it will be zero when there is one single label. Thus, it tends to zero when the dataset is very unbalanced. In our verification routine, we use by default the minimum threshold of 0.5 to flag the labels data as unbalanced. This will be reported to the user as warning and will affect the verification of the initial bias.

### Starting DNN state

Starting from different initial states, the optimization algorithm follows different trajectories and can terminate at different equilibrium states. Thus, a poor initial state adversely affects the optimization routines, and consequently, the optimality of the equilibrium state.

**Initial Weights Verification** First, we verify that there are substantial differences between the parameter's values by computing the variance of each parameter's values and checking if it is not equal to 0. Next, one can make sure that, given the chosen activation function, the distribution of initial random values are sampled from a uniform or normal distribution with a careful tweak, i.e., by calibrating attentively the variance because the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. The equality between the actual variance of each weight and its recommended variance given the input size is verified using f-test [200]. In the following, we describe the recommended variances depending on the activation function of the corresponding layer.

- LeCun [198] proposes a heuristic that initializes each neuron's weight as either $\mathcal{N}(0, \sqrt{1/fan_{in}})$, i.e., normal distribution with zero mean and $1/fan_{in}$ of variance or $\mathcal{U}(-\sqrt{3/fan_{in}}, +\sqrt{3/fan_{in}})$, i.e., uniform distribution within $[-limit, limit]$ and $limit = \sqrt{3/fan_{in}}$, where $fan_{in}$ is the number of inputs. This guarantees that all the initial neurons' weights have approximately the same output distribution, and its empirical evaluation on sigmoid layer activation shows a significant improvement on the rate of convergence.

- Glorot et al. [24] recommend the following initialization (especially when tanh is used as activation function) which consists of neuron's initialization following

either $\mathcal{N}(0, \sqrt{2/(fan_{in} + fan_{out})})$ or $\mathcal{U}(-\sqrt{6/(fan_{in} + fan_{out})}, +\sqrt{6/(fan_{in} + fan_{out})})$, where $fan_{in}$, $fan_{out}$ are the number of inputs and outputs.

- He et al. [84] also proposed an initialization specifically for ReLU neurons. They suggest that the variance of neurons in the network should be $2/fan_{in}$, which gives an initialization of either $\mathcal{N}(0, \sqrt{2/fan_{in}})$ or $\mathcal{U}(-\sqrt{6/fan_{in}}, +\sqrt{6/fan_{in}})$. All of the above initializations have been discovered empirically and proven to be optimal in classic well-known CNNs like LeNet [163] and modern architectures such as VGG [151] and GoogleNet [201].

**Initial Biases Verification** As a baseline, we verify that the bias exists, and initially is set to 0 in case of well distributed labels. Nevertheless, we proceed by more advanced check on the last bias initialization in case the pre-check on unbalanced labels for classification was fired. Indeed, we make sure that the bias set for the output layer reflects the bias already found in the distribution of outcomes in the given ground truth data. In our implemented verification routine, we consider the unbalanced classification problem, where it is usually effective to set each bias unit $b_i$ to $\log(p_i/1 - p_i)$, where $p_i$ is the proportion of training instances of the label corresponding to the bias $b_i$ of unit $i$ [202]. Concerning the regression problem, if the coefficient of variation w.r.t each output $j$ (i.e., the ratio of the standard deviation to the mean) is low (e.g., our default threshold is 0.1%), we verify that its corresponding $b_j$ is set to $m_j$, the mean value of the supervised target $j$. This eases the optimization by transforming the regression problem into predicting the deviation against the baseline (i.e., the mean value).

**Cold Start Loss Verification** An unexpected loss at the iteration 0 with an untrained model (cold start), can indicate numerous issues including faulty loss function, ineffective loss reduction strategy, as well as buggy parameters' initializers [203]. Indeed, we compute the loss at cold start with an increasing size of batches in order to verify that the obtained losses are not proportionally increasing. This validates that the loss estimation is an average-based expectation and it is not a sum-based reduction. In our verification routine, we duplicate a random batch of data by doubling its size ($\times 2$), then, we check if their corresponding losses at cold start are doubling ($\times 2$) as well. Next, we verify that the optimizer starts well at the expected initial loss (i.e., the one estimated at the first run with random internal parameters). It is always possible to derive approximately the correct initial loss for a given DNN program configuration. For instance, the cross-entropy loss for a balanced classification problem should start with uniformly distributed probabilities of $p(label) = 1/L$ and initial value of $loss = -\log(1/L)$, where $L$ is the number of target labels.

**Tensor-Based Operation Verification** A careless developer can introduce mistakes when transforming math formulas or pseudo-code from white papers to tensor-based operations using basic DL libraries calculation APIs. As the DNN tensor-based computations include intensive broadcasting and reduction operations in order to perform individually calculation over all the instances/neurons at each level, then, reduce the calculations to define aggregative scores towards summarize all into a scalar cost that represent how well the DNN performs on the data. We found that common implementation mistakes miss or add unnecessary dependencies to network components (instances, neurons, scores,..etc). Thus, it is important to test that a written operation depends only on its related components. For example, an activation function should be applied separately on all the neurons, which means the activation output $i$ should depend only on the neuron input $i$, or a distance calculation between prediction and actual values should contain independent components, where the distance value $i$ depends on only the prediction $i$ and the ground truth label $i$ as well. To validate the correctness of dependencies of all the math operations for a computed quantity within the neural network, we use the gradients flowing in the network to debug the dependencies between each operation's components given the fact that the gradient of a function w.r.t an independent component is always zero [180]. For instance, let's consider a newly-implemented loss function, we can extract only the loss obtained for the outputs of a data point $i$, and then perform a full backward pass to the input data in order to make sure that only the gradient w.r.t the $i$-th input data is not null. A violation of this condition signals that overlapping dependencies exist, which means that the on-watch average loss for DNN performance is wrong and misleading.

**Computed Gradient Verification** The backbone of backpropagation implementation lies in the computation of gradients with respect to different DNN operations, including linear weighted sum and non-linear activations. Whenever DL developers add hand-crafted math operations and gradient estimators, we perform a numerical gradient checking that consists of comparing between the analytic and numerical calculated gradients, respectively, the gradient produced by the analytic formula and the centered finite difference approximation, $\frac{f(x+h)-f(x-h)}{2h}$. Both gradients should be equivalent, approximately equal, for the same data points. The following steps are recommended by [25, 203] to improve the effectiveness of this gradient checking process (and the detection of faulty gradients).

***Relative error comparison:*** The difference between numerical gradient $f'_n$ and analytic gradient $f'_a$ represents the absolute error that should not be above a predefined threshold. However, it is hard to fix a common threshold of absolute error for DNN because its internal computations are usually composed of multiple functions; so the errors build up through backpropagation. Thus, it is preferred to use a relative error, $\frac{|f'_a-f'_n|}{max(|f'_a|,|f'_n|)}$, that might be

acceptable below $1e^{-2}$.

***Sampled data instances:*** Sampling a few data instances for numerical gradient checking reduces the risk of crossing kinks, which are non-differentiable areas of the loss landscape. For instance, ReLU has zero gradient at the origin; but the numerical gradient can cross over the kink and produce a value different from zero. Besides, DNN's parameters are large with thousands of dimensions; so the computation error could be on a random subset of each gradient dimension. Therefore, a random sampling among both data points and dimensions makes the finite-difference approximations less error-prone and faster in practice.

***No regularization:*** The standard regularization can render large errors, misleading the numerical gradient checking when the penalty term added overwhelms the original loss (i.e., the gradient is mostly related to penalty cost). Moreover, advanced regularization techniques such as dropout induce non-determinism in the DNN internal computations, which enhances the error-proneness of numerical gradient checks.

***Prior burn-in training:*** A short burn-in training during which the parameters take better and more representative values than randomly initialized ones is recommended. It is also recommended to avoid the gradient checking at cold start since it could introduce pathological edge cases, masking a buggy implemented gradient.

**Fitness of a single batch of data**

Given a tiny sample of data, the target problem becomes easy to solve and the training algorithm should be able to converge to a DNN that fits the data without any issue, as every well-designed DNN should be able to fit a small dataset. This is a main pre-check for DNN training routines because it is a necessary condition, where its non-satisfaction indicates a misconfiguration or a software bug.

**Input Dependency Verification:** We confirm that training programs on zeroed batches of data perform worse than those on real samples of data. This check was initially proposed by Karpathy [180], as a verification that the model outperforms an input-independent baseline. For debugging, this improvement over the input-independent baseline shows that the training program is successfully leveraging the input to optimize the DNN parameters.

**Overfit Verification:** We verify that the optimization mechanism is working well on a controlled sample of data with reduced size (i.e., a few data points for each class) [25, 203]. The acceptance criteria is that the DNN achieves 100% accuracy or near-to-zero absolute errors (AEs) on continuous outputs (by default, we consider AE in the order of $10^{-3}$). A

failure to achieve this performance would signal the presence of issues regarding the DNN optimization routines.

**Regularization Verification:** As above-mentioned, the controlled experiment of fitting a single batch would lead to overfit the provided few data points in normal situations, however, the loss should be greater than zero [25] when there is quite regularization. This check can be improved by watching furthermore the smoothness of the loss curve to spot a lack of noise in the optimization, and consequently, it reinforces doubts about the absence of active regulation. In our verification routine, we set up $1e-5$ and 0.95 as default thresholds for loss and smoothness ratio 5.2.4 to recognize suspicious loss curves that are smoothly decreasing towards a very low loss. This shows the model's propensity to overfit quickly the training data caused by a lack of regularization, which often implies high risks of capturing useless residual variations in the given features.

Regarding pre-check of a single batch of data fitness, we concentrate on the functioning of the training algorithm, and precisely, its ability to converge (i.e., reaching an optimal equilibrium state) given a reduced size problem. However, the DNN training may still contain inefficiencies that did not prevent it from solving a small problem size but would affect its performance when it trains on larger problem size. Moreover, the failure at these batch fitness prechecks do not provide indications about the possible root causes behind this incapacity to successfully pass them. Therefore, the next phase in our debugging process aims to guarantee the "proper functioning" of the training algorithm given a reduced size problem. What we mean by "proper functioning" is the ability to converge with a valid accomplished trajectory (i.e., passing through valid intermediary DNN states).

### 5.2.4 Phase 2: Proper fitting a single batch of data

At this phase, a monitored training is launched on a representative sample of data (i.e., a single batch of data resulting from a stratified sampling). The monitoring routines serve to detect early and precisely the potential issues with diverse automated verifications that watch periodically for the DNN components' misbehaviors to spot and report properties' violations. In case of a DNN with major issues, i.e., already failed the precheck on single batch fitness, this phase allows to steer the user towards problematic components and provide meaningful messages on the violated properties that restrict further the potential root causes. Otherwise, DL practitioners can still leverage this debugging phase for pre-examined DNNs to spot inefficiencies of their design choices that cause training instability in regards to learning updates or activation distributions, which might prevent the DNN from reaching optimal

steady states and lead to a wastage of time and resources on unnecessary long training sessions with the full datasets.

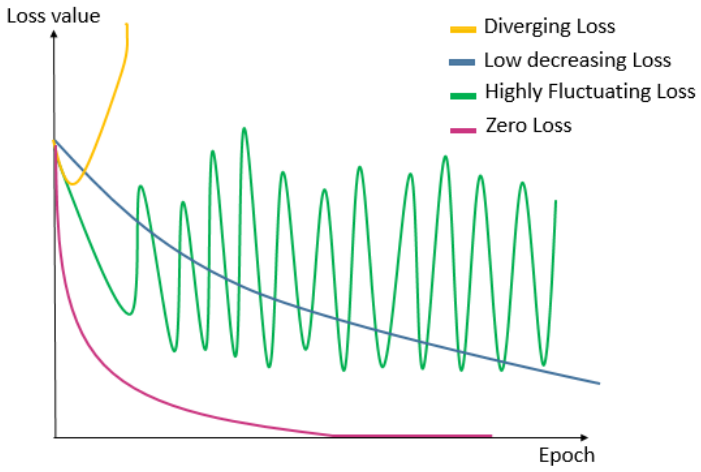



Figure 5.5 Illustration of loss minimization issues

### Abnormal Loss Curvature

A loss curve is a plot of model loss value over time in terms of epochs or iterations. The shape and dynamics of a loss curve are useful to diagnose the behavior of the optimizer against the target learning problem. Figure 5.5 shows the abnormal loss curves, detailed-below, which indicate different pathologies of statistical learning from data. The anomalous loss evolution [203] can be detected using continuously updated metrics that are cheap to compute and which can reveal anomalies effectively.

**Non- or Slow-Decreasing loss** A flat or low slope down curve shows that the loss is either non-decreasing or decreasing very slowly which means that the model is not able to learn at all or has a low learning capacity. This could be due to an inadequate loss function or a low learning rate. As introduced in 5.2.2 for the stagnation test, we verify that the loss is decreasing at an acceptable decay rate for a window of steps, e.g., we set up 5 steps by default. For the loss decreasing verification, we proceed by watching that the percentage difference between consecutive losses' values, $loss\_decay\_ratio = \frac{previous\_loss - current\_loss}{previous\_loss}$, is greater or equal to a predefined minimum percentage difference (by default, we set up 5%) by which the loss should decrease, $loss\_decay\_ratio > prefixed\_min\_loss$.

**Diverging loss** A curve with a high slope indicates that the loss is diverging with wildly increasing values which means that the optimization problem is turned into a loss maximization

instead of minimization. This could be the result of a high learning rate or a buggy gradient. Therefore, we verify that the loss has no increasing tendency instead of decreasing one. This verification can be done automatically by updating a reference minimum loss ($lowest\_loss$), and then, watching that the absolute ratio of loss, $abs\_loss\_ratio = \frac{current\_loss}{lowest\_loss}$ (i.e., we call it absolute ratio because it is computed w.r.t the lowest obtained loss) is not dramatically diverging, as described in 5.2.2, during the training.

**Highly-Fluctuating loss** A noisy curve with random fluctuations demonstrates that the loss is not converging to a line of stability, hence, the optimization is facing difficulties preventing it from converging normally. Potential reasons behind this issue could be: strong regularization that gives rise to noisy loss estimation, or high learning rate that produces large updates keeping the optimizer jumping over the local minimum without converging. Hence, we compute the smoothness ratio of the loss curve as follows:

$$smoothness\_ratio = \frac{N\_samples - N\_direction\_changes}{N\_samples}$$

Where $N\_samples$, $N\_direction\_changes$ denote, respectively, the sampled iterations count and the number of alternations in the loss evolving direction. Then, we check periodically that the smoothness ratio is not lower than a predefined threshold, otherwise, it indicates a high amount of fluctuations (e.g., we fix 0.5 as a default threshold, which means that more than 50% of consecutive sampled losses are having altered directions).

### Performance Metrics Correlation

DL practitioners should select or implement the right loss measure (i.e., mean squared error, cross-entropy, etc.) that will be set as objective function for parameters optimization, while they keep watching a target performance metric (i.e., R-squared, Accuracy, etc.). For this verification routine, we compute, continuously, the absolute value of correlation coefficient between the optimized loss and the target performance measurements between training steps. The latter describes the magnitude of the relationship between two variables within the interval of $[0, 1]$ and should not become lower than a predefined threshold (e.g., our verification routine reports an absolute correlation coefficient of less than 0.5). This provides an indicative metric of how representative the optimized loss is with respect to the target objective and vice-versa. Thus, a poor choice of loss function would have high chances to be uncorrelated with the performance metric, similarly, a buggy performance calculation would yield incorrect values with low correlation with the loss evolution.

### Unstable Gradient

We propose to detect unstable gradient issues by examining, continuously, the evolution of estimated gradient's values with respect to each DNN's layer. More specifically, we check the growth and decay rate of the absolute average of the layer's gradients, estimated for the last iterations, to detect if the latter is following an unstable evolution trend, i.e., it is exploding or diminishing, as described in 5.2.2.

### Magnitude of Regularization Penalty Cost

In addition, there is an issue with the addition of a high amount of regularization to the computed loss, whether by using a high *lambda* value or a poorly-designed penalty regularization cost. Indeed, Park et al. [204] highlight that the learning suddenly fails when the magnitude of gradients from $loss(W, b, D)$ decreases faster than that from $\Omega(W)$; so the penalty term gradient overwhelm the loss data gradient. In such problematic situations, the weights' updates become mainly related to the regularization term, which causes the failure of the model to learn.
One should ensure that the regularization is not too strong. The regularization term gradient should not dominate and suppress the loss gradients w.r.t the weights.To ensure this, we recommend watching continuously, the proportion of the magnitude of the penalty terms' gradients w.r.t the magnitude of loss data in order to validate that it is not diverging, as described in 5.2.2.

### Parameters States and Dynamics

The main DNN's on-training parameters, weights and biases, should be optimized towards better solving the learning problem over the training sessions. In the following, we propose different verification routines on the parameters current states and update dynamics that could indicate convergence issues and non-optimality of the on-going estimation.

**On-Training Parameters Verification** A preliminary sanity check for neural network parameters would be verifying their changing estimations over the training iterations. We make sure that all defined trainable layers have their parameters updated. Indeed, a non-zero difference between the values of trainable parameters before and after the execution of a few training passes (i.e., optimization updates) confirms that the dependencies between the layers are correctly set up and that all the trainable parameters are getting optimized.

**Dead and Over-Negative Weights Verification** A layer's weights are considered dead or over-negative, when respectively, the ratio of zeros or negative values in the tensor elements is very high [187]. These two states of weights are likely to be problematic for the learning dynamics. Indeed, given a common DNN setup (e.g., normalized inputs within $[0, 1]$ and a variant of ReLU as hidden layers activations), null or negative learned weights within hidden layers represent connections to intermediate features that do not contain any relevant information for the target task. Thus, if a layer's weights are mostly full of null or negative weights, their corresponding activation layers are likely to stagnate on a non-optimal flat region and consequently, the DNN would start facing dead layers (i.e., ReLUs mostly outputting the value zero) and frozen layers (i.e., no updates of the weights). In our verification routine, we flag any tensor of layer's weight ($W$), having either the ratio of very low values (i.e., lower than $1e-5$) or negative values is higher than a predefined threshold (i.e., 95% is used by default), as respectively, dead or over-negative weights.

**Stable Parameters Update Verification** Deep neural networks introduce challenges regarding learning stability compared to shallow networks. In the training pitfalls section 5.1, we discussed some practices such as tuning the learning rate and adding activation or weight normalization to balance out the learning speeds for the hidden layers. Thus, it is important to make sure that the parameters' updates are stable. Hinton [202] and Bottou [205] proposed the following heuristic, the magnitude of parameter updates over batches should represent, respectively, 0.1% or 1%, of the magnitude of the parameter itself, not 50% or 0.001%. Therefore, we propose a verification routine to detect unstable learning parameters by comparing the magnitude of parameters' gradients to the magnitude of the parameters themselves. More specifically, following the recommendation to keep the parameter update ratio around 0.01 or 0.001 (*i.e.,* $-2$ or $-3$ on base 10 logarithm), we compute the ratio of absolute average magnitudes of these values and verify that this ratio doesn't diverge significantly from the following predefined thresholds:

$$-4 < \log_{10}\left(\overline{|W^{(i+1)} - W^{(i)}|}/\overline{|W^{(i)}|}\right) < -1$$

The proposed verification mechanism reports irrelevant layers (i.e., where updates are unstable) and frozen layers (i.e., where updates are stalled) to the user.

**Parameters Diverging Verification** Worse than unstable learning, weights and biases risk divergence, and may go towards $+/-\infty$. For instance, high values of initial weights or learning rate with a lack of–or–insufficient regularization provokes highly-increasing weights'

updates, leading to bigger and bigger values, until reaching $\infty$ (this is caused by overflow rounding precision). In addition to that, biases can also become huge in certain situations where features could not explain enough the predicted outcome or might not be useful in differentiating between the classes. Therefore, we automate a verification routine that watches continuously the absolute averages of parameters are not diverging, as described in 5.2.2.

### Activations Distribution

**Out-of-Range Activation Verification** Given a newly-implemented activation function, we include a baseline verification routine to watch that the produced activations are within the expected range of values. This would be useful to find computation mistakes or misconceptions causing out-of-range outputs.

**Validation of Output Activation Domain** The last layer's activation represents the outputs of the neural network, which should produce valid outcomes while covering the whole distribution of the possible ground truth labels. We implement a verification routine to check that the outputs of classifiers are probabilities, i.e., positive values within $[0, 1]$ and sum-to-one in case of multidimensional output. For regression, we empirically verify that the predicted outputs over the iterations were able to satisfy necessary conditions derived from ground truth boundaries, i.e., non-zero variance, can be negative, can exceed 1.0. Under these conditions, the common identified faults of using the wrong activation function can be detected.

**Saturated Bounded Activation Detection** To detect saturation issues in DNNs, we compute single-valued saturation measure $\rho_B$ proposed by Rakitianskaia and Engelbrecht [206] if the hidden activations are bounded functions such as sigmoid or tanh. This measure is computed using the outputs of an activation function and is applicable to all bounded activation functions. It is independent of the activation function output range and allows a direct statistical measuring of the degree of saturation between NNs. $\rho_B$ is bounded and easy to interpret: it tends to 1 as the degree of saturation increases and tends to zero otherwise. It contains a single tunable parameter, the number of bins $B$ that converges for $B \geq 10$, *i.e.,* it means splitting the interval of activation outputs into $B$ equal sub-intervals. Thus, $B = 10$ can be used without any further tuning. Given a bounded activation function $g$, $\rho_B$ is computed as the weighted mean presented in Equation 5.2.

$$\rho_B = \frac{\sum_{b=1}^{B} |\bar{g}'_b| N_B}{\sum_{b=1}^{B} N_B} \tag{5.2}$$

Where, $B$ is the total number of bins, $\bar{g}'_b$ is the scaled average of output values in the bin $b$ within the range $[-1, 1]$, $N_b$ is the number of outputs in bin $b$. Indeed, this weighted mean formula turns into a simple arithmetic mean when all weights are equal. Thus, if $\bar{g}'_b$ is uniformly distributed in $[-1, 1]$, the value of $\rho_B$ will be close to 0.5, since absolute activation values are considered, thus all $\bar{g}'_b$ values are squashed to the $[0, 1]$ interval. For a normal distribution of $\bar{g}'_b$, the value of $\rho_B$ will be smaller than 0.5. The higher the asymptotic frequencies of $\bar{g}'_b$, the closer $\rho_B$ will be to 1.
This verification routine can be automated by storing for each neuron its last $O$ outputs' values in a buffer of a limited size. Then, it proceeds by computing its $\rho_B$ metric based on those recent outputs. If the neuron corresponding value tends to be 1 (e.g., a threshold of 0.95 is used in practice to spot this tendency), the neuron can be considered as saturated. After checking all neurons for saturation, we compute the ratio of saturated neurons per layer to alert the DL developers about layers with saturation ratios that surpass a predefined threshold (e.g., we fix 50% by default).

**Dead ReLU Activation Detection** By definition, a given neuron is considered to be dead if it always returns zero [207]. Hence, we detect practically dead ReLUs by reducing the last outputs for each neuron stored in the limited size buffer into a single $95^{th}$ percentile, which is more robust than the maximum reduction against outliers (e.g., a non-zero stored values from earlier training iterations). Thus, we mark all the neurons with a $95^{th}$ percentile less than a predefined threshold (i.e., by default, we set up $1e-5$). Next, we proceed by calculating the ratio of the marked dead neurons per layer and we flag the layers with a number of dead neurons higher than a predefined threshold (e.g., we fix a default threshold of 50%).

**Unstable Activation Detection** Although this unstable activation issue is more generic than dead or saturation phenomena, DL experts usually watch the histograms of sampled activations from each layer while expecting to have normally-distributed values with unit standard deviation, e.g., a value within $[0.5, 2]$ has been shown to be an acceptable variance of activations [30]. Thus, we base on this expert's heuristic to statistically validate that the sampled activations of each layer over the last iterations is having a well-calibrated variance scale. Concretely, the test would pass the actual standard deviation ($\sigma_{act}$) belongs to the range of $[0.5, 2]$; otherwise, we perform an f-test [200] to compare $\sigma_{act}$ with either the low-bound 0.5 if $\sigma_{act} < 0.5$ or 2.0 if $\sigma_{act} > 2.0$.

DL practitioners can perform a closed feedback loop using this inexpensive and rapid debugging on a single batch of data until fixing all the covered coding bugs and improving the

settings in a way that enhances the chances of converging to a more optimal model. Thus, a successful pass at this phase increases the confidence that the training program is devoid of common DNN pathologies such as: vanishing gradients, saturation of activation functions, and inappropriate learning speed. Nevertheless, other components of the training program have not been tested yet. For instance, a training program may pass all the single batch debugging checks, while the data loader can inject too much noise or mismatch features and labels, which yield corrupted batches of data, and consequently, the resulting DNN does not solve the target problem.

### 5.2.5 Phase 3: Post-fitting conditions

Once the fitting of a single batch step is successfully passed, several post-fitting conditions should be satisfied to guarantee the correctness of the data loader, the data augmentation module, and the advanced regularization techniques that require additional computations during the inference. The following debugging phase validates the behavior of the DNN training program during regular training sessions, i.e., we use the available training and validation data.

#### Distribution-Shifting Augmentation Verification

We propose a post-check that verifies the validity of the augmentation methods, applying data transformations on the generated batches to enhance diversity and improve the generalizability by smoothing the loss landscape and forcing the model to capture the invariants useful for the target task. A valid augmenter should not introduce a shift in the data distribution that makes the model perform worse on the original dataset. For instance, overwhelming noise injection leads to produce meaningless inputs; so both of ratio and scale of the injected noise should be carefully picked to hold the augmented data distribution close to the original one.

To detect this poor design of data transformations, we debug the data augmenter module by comparing the performance and the activation patterns of the DNN trained with-and-without augmentation on a sample of data from the validation set. Concerning the measurement of dissimilarity between the activation patterns, we use Centered Kernel Alignment (CKA) [208], which is an optimized and powerful representational similarity measure, allowing the assessment of the differences and the correspondences between patterns learned by different DNNs or same DNN with different data. Indeed, given two matrices $X$,$Y$, where $X \in \mathbb{R}^{m \times n_1}$ is a matrix of activations of $n_1$ neurons for $m$ examples and $Y \in \mathbb{R}^{m \times n_2}$ is a matrix of activations of $n_2$ neurons for the same $m$ examples, a DNN representational similarity

index $s(X, Y)$ estimates the similarity between the representations learned in both matrices of activations. Thus, the CKA's empirical validation [208] shows its ability to compare representations within and across DNNs, in order to assess the effect of changing variation factors on the DNN training. This similarity metric is robust and it does not affect by the stochasticity of the optimizer or the random initializations, which makes it suitable for our verification on the resulting activation patterns instability when the augmented data is noisy and shifted w.r.t the training data distribution.
Therefore, the DNN training program would fail the test if there is a degradation of the performance and a substantial difference in the activation patterns between the two trained DNNs. In our implementation, we check for an increase in the loss ($loss_{augm\_data} > loss_{orig_data}$) and a decrease of activation pattern similarity (e.g., we set up a threshold of a minimum CKA index equals to 0.8, which means a decrease of 20%). Indeed, any difference in the activation pattern could be acceptable and might be considered as an improvement in the detected patterns in case the performance is enhanced, however, having both of the activations pattern divergence and the performance degradation indicate strongly that the augmented data induce a distribution shift.

### Corrupted Labels

Given a corrupted data shuffler, the paired data (features $X$, labels $y$) are mismatched, where the row label $y_i$ does not correspond to the row features $X_i$. Since the shuffling is executed after each epoch (i.e., full pass over the data), a corrupted data loader will generate a new distribution of supervised training data every time it is called. Thus, the training loss curve would be subject to intermittent spikes because the neural network starts learning on a new distribution at the 1th-iteration of every epoch. Based on this observation, we propose to collect all the 1th-iteration losses into a set and perform a statistical test to detect if there is significative difference between them, which means the loss estimated on the first sampled batch of data is improved over the epochs; otherwise, we flag the data loader as corrupted because the DNN successfully passes all the previous verifications, but cannot improve its performance over the epochs; so it is high probable that the data loader is falsifying the batches of data in-between the elements' shuffles.

### Unstable Mode Transfer: From Train to Inference Mode

Advanced regularization techniques like dropout or batchnorm introduce, respectively, noise-injection and normalization mechanisms in order to grant the DNN training access to sub-model ensembling and well-conditioned loss minimization. They incorporate two functional

modes: the training mode and the inference mode; so many bugs can remain silent and hidden during the training mode, but the transfer to the inference mode can reveal them through DNN misbehaviors and divergences induced by the mode transfer [177, 178]. Thus, we construct a verification routine that detects the behavioral shift occurring when transferring the DNN from the training mode to the inference mode. Our implemented behavioral difference assessment is based on the CKA metric for representational dissimilarity between different modes' activations (e.g., we fix a default threshold of 0.75 as minimum similarity of activation patterns on the same data), and the relative change between the modes' losses (e.g., the threshold is by default set to 50%). Then, one can check the different regularizers' internals, including *pkeep*, $E[x]$ and $Var[x]$ to diagnose the root cause.

### 5.2.6 TensorFlow-based Implementation

To assess the effectiveness of our proposed debugging approach on real faults in DL-based software systems, we implemented a TF-based library that performs the debugging phases on a training program written using TF features. Indeed, we choose to focus on TF-based training programs because of the popularity of TF in the ML community [209]. Nevertheless, the property-based approach proposed in this paper can be adapted for other DL frameworks. In the following, we describe the components of this TF library.

#### Setting Up the Testing Session

The testing of a DNN training program can be more complicated than for a traditional program, because of its non-deterministic aspect. It is difficult to investigate the training issues and identify the root causes when the program exhibits a substantially new behavior for each execution. To reduce the stochasticity of a DNN training program, we fix all the random seeds of all the computational libraries beforehand, which guarantees the reproduction of the same random variables. Furthermore, we offer the option of deactivating the parallelism, if a tester wants to obtain a perfectly reproducible result. As default settings, we allow leveraging multi-cores CPUs and GPUs through multithreading execution. We allow this because a single-thread execution slows down dramatically the training time. Also, *TheDeepChecker* targets major training issues related to erroneous training behaviors caused by the introduction of coding bugs or misconfigurations, which differ from minor training issues caused by non-optimal choices of hyperparameters, leading to near-to-optimal DNNs instead of the best-fitted one. Therefore, the resulting pathologies in the training dynamics would likely be persistent to the possible low magnitude differences between multiple executions of parallel floating-point computations.

### Fetching and Monitoring the Training Program Internals

TFDBG [210] is the official debugging tool specialized for TF programs that offers features such as inspection of the computational graph, addition of conditional breakpoints and real-time view on internal tensor values of running TF computational graphs. However, it is not practical for our approach since it adds a huge overhead on computation time, as it handles each execution step of the graph, to allow debugging the issues and pinpoint the exact graph nodes where a problem first surfaced. In fact, the implemented verifications fetch the values of tensors, representing parameters, activations, and gradients, by requesting through the provided API the on-running DAG that encodes both the DNN design and the used training method. The routines that continuously monitor the state of the neural network do not need to break neither the feed-forward nor the backward passes since they can access the internals of the intermediate neural networks to detect pathologies in the learning dynamics. This allows the monitored training iteration to be executed in an atomic way and avoid the overhead of using TFDBG. To manage our set of verification routines running simultaneously, we use the monitored session and hooks mechanism; to handle the additional processing injected between training iterations. To do that, we need to perform the following two steps:

1. Create one or more Hooks objects that implement methods such *before_run* and *after_run* to access the intermediate tensors' values of activations, parameters, and gradients, then, apply the verification logic.

2. Create a *Monitored Session* that handles the execution of hooks' additional treatments before and after running each training pass.

### Buffering the DNN's status data

The activations and gradients represent intermediary computations over, respectively, forward and backward passes. The weights and biases represent learnable parameters that are updated, continuously. As a result, the internal tensors do not survive between two consecutive training iterations. Thus, we implement buffer data storage to save incrementally the values of watched tensors. By default, we set the size of the buffer to 10 elements but we keep it a configurable option in the debugging tool.

### Running the checks

Conceptually, each check performs the following two steps. First, the instantiation of the property requires the computation of necessary metrics in relation to the targeted violations.

Regarding the on-training verification routines, a data preparation using a reduction strategy (i.e., average, norm, and quartiles, etc.) is necessary to aggregate the accumulated tensors in the buffer data storage. Second, the issue detection consists of applying a heuristic-based verification rule, involving the inferred metrics and predefined thresholds, that captures the negative effects and anomalous training behaviors induced by the targeted issue. Indeed, the choice of thresholds would affect the sensitivity and specificity of the issue detection. Additionally, the training pathologies are correlated to each other and a particular bug can be the root cause of multiple of them. Thus, our dynamic debugging strategy alleviates these challenges by leveraging limited-size buffers, continuous verification routines, and informative raised errors. Therefore, *TheDeepChecker* implements a debugging feedback loop that does not stop after finding violations, but it keeps watching the training execution while dynamically producing error messages that steer the DL developer to further narrow down the possible root causes and avoid errors conflicts relying on the persistence of the errors, the chronological order of the raised issues, and the reported information.

**Shrinking the suspicious state and raising errors**

Once an issue is detected, *TheDeepChecker* shrinks the state of the on-training DNN to communicate the component where the property violation is found and its corresponding indicators including the reference of the layer, the computed metric, and the predefined threshold. In fact, the reported information should provide an explanation of why the underlying property is considered to be in violation. First, reporting the shrinked neural networks' states helps the user avoid false positives. For instance, the learning speed can start by relatively high updates that can be close or-even slightly larger than the prefixed update magnitude threshold. Thus, a single warning message including the current update magnitude and the surpassed threshold could help the user decide whether or not the unstable learning issue occurs in the current situation. Second, the shrinking of the buggy DNN state is useful to pinpoint the suspected computational units that developers should investigate, and therefore, it helps them identify the occurred issue's root cause. For example, the computational layers send information throughout the DNN by using edges that connect layers to one another during the forward pass and they receive updates from the gradients of the loss flowing reversely over the same edges during the backward. Thus, the level of the dead layer and its amount of dead neurons, as well as, the level of vanishing gradient and its magnitude scale can be used to identify the actual unstable layer, where the information is no longer flowing during either forward or backward pass. Developers would first investigate the underlying layer and then fix the bug within the program using this information.

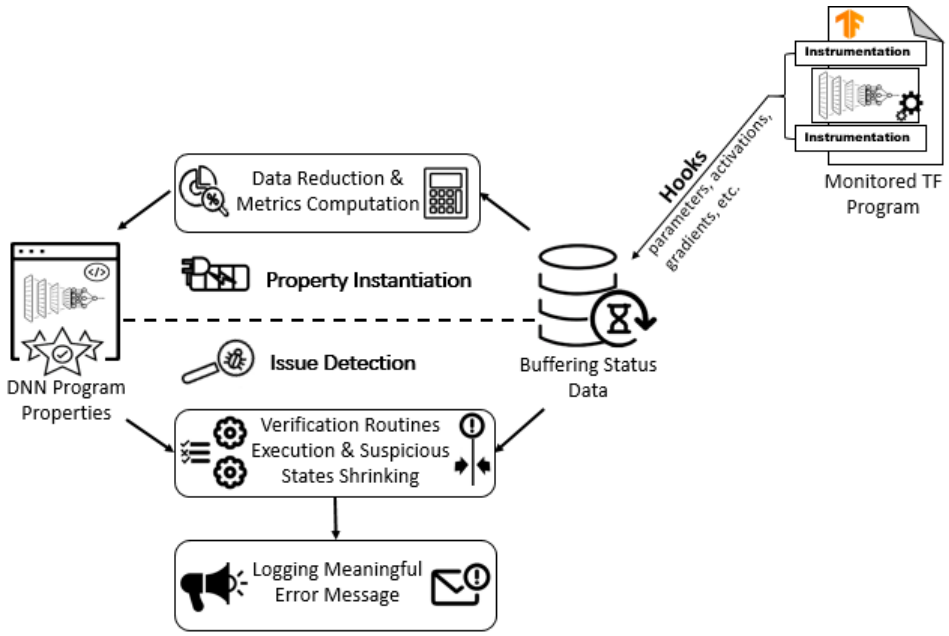



Figure 5.6 Illustration of our Property-based Debugging Approach for TF Programs

Figure 5.6 summarizes the above-mentioned implemented steps of our property-based debugging approach for TF Programs.

## 5.3 Evaluation

The objective of this evaluation is to assess the effectiveness of our proposed property-based debugging method in allowing for the early detection of real bugs that occur in DNN-based software systems. We also conducted a usability study with two professional DL engineers to assess the relevance of *TheDeepChecker*'s error messages at guiding developers in identifying the root cause of bugs and fixing them.

### 5.3.1 Design of Case Studies

In this section, we describe the design of our case study that aimed to assess the performance of *TheDeepChecker*. We explain how we selected and reproduced relevant bugs for our evaluation.

#### Real Faults in DNN-based Training Programs

The reproduction of buggy DNN training programs is quite difficult because of the rapid evolution of TF API functions and even infeasible when major code blocks, datasets, or

environment settings are missing. In previous research works, program mutation [45] was leveraged to evaluate the quality of DL program debugging approaches. The mutation relies on well-defined rules to change slightly the syntactic structure of the code, or mimic systematically application-specific errors. Then, the debugger should detect and reject mutants, which is called killing the mutant in such analysis. Thus, its effectiveness is measured by the ratio of killed mutants w.r.t the total of generated mutants. Xie et al. [92] leverage MuJava [211], which is an automatic java code mutator, to produce defective mutants of Weka's implementations [212] of $k$-Nearest Neighbors (kNN) and Naive Bayesian (NB). Dwarakanath et al. [93] use MutPy [213], an open-source tool for python code mutation, to inject typical programming errors in a clean implementation of deep residual neural networks (ResNET). Both mutation analyses rely on language-specific code mutators that alter randomly the arithmetic operators, logical operators, variable's scope and casting types, etc. However, even if these random code alterations mimic a large variety of coding mistakes, they cannot be representative for the real-world buggy training programs, where the code is relatively short with heavy dependency on tensor-based computational libraries. Besides, Ma et al. [94] proposed DeepMutation that defines DL-specific mutation rules for DL programs, including a layer addition, a layer removal, and an activation function removal. Based on our investigation on DL faults, we found that these operators can actually mimic real faults in relation to missing DNN components, such as missing batchnorms (i.e., removing the normalization layer) or redundant softmax (i.e., adding another softmax on the output layer). However, many of the generated mutants using these operators can be equivalent to the original network, or even better for solving the underlying learning problem. This makes the evaluation based on the ratio of killed mutants misleading. Therefore, it is necessary to assess the effectiveness of our debugging approach on detecting the real DL bugs that have been experienced and reported by the DL practitioners. As shown in Figure 5.1, we have collected concrete instances of real DL bugs [1] that cause training issues without crashing the DL program. Thus, we rely on their identified root causes and symptoms to inject each DL fault into clean DL programs to force the creation of valuable synthetic buggy versions. In the following, we detail the two high-level categories of common root causes for the non-crashing DL bugs.

**Coding Bugs in DNN-based Training Programs**

Like any software system, DNN training programs may contain the missing and wrong code statements that cause a deviation between the designed DNN and its corresponding written code (see Table 5.1). In fact, the DNN training program is implemented using conventional programming languages, which may include coding faults. The lack of oracle for internal variables and the stochastic nature of the DNN optimization process, make most of these

coding mistakes hidden without disrupting the flow of the program's execution. For instance, the majority of these hidden bugs are incorrect math operations such as flipped sign result, inverted order of calculations, wrong data transformations. We found another type of coding bugs in TF programs that consist of a TF API misuse committed by developers who misunderstood the implicit assumptions regarding how to use these configurable routines. In fact, modern DL libraries provide rich APIs covering more and more state-of-the-art techniques. Consequently, the API routines integrate more and more configurable options and set up default values to make them ready-to-use for quick prototyping. However, they assume that their users are capable of configuring them properly, which is unfortunately not always the case. Some of the built-in data loaders, for example, automatically perform standard preprocessing of numerical data, such as normalization. This misleads some rookies to blindly perform a double linear scaling afterward. Another common API misuse is related to the recent versions of loss functions. Indeed, numerically stable implementations regarding some of the loss functions require merging the loss and output activation formula together to rewrite them carefully without any potential $\log(0)$ or $\exp(\infty)$. However, users may ignore this gap between theoretical loss function and built-in numerical stable ones; which may result in redundant activations.

**Misconfigurations of DNN-based Training Programs**

Modern DNN training programs are highly-configurable software built using routines from DL libraries. Their correct settings, given the context, becomes a challenging task and if an incident occurs due to misconfiguration, the on-training DNN may produce misleading performance faults. These misconfigurations assemble all the wrong and poor choices for the configuration of DNN-based software systems, including the DNN design and the training method (see Table 5.1). The lack of understanding of DL fundamentals is the main reason behind the occurrence of these configuration issues, especially, when dealing with a novel DL technique or facing an unfamiliar target problem. For instance, numerous misconfigurations in relation with random initializers, loss functions, normalization methods and optimization hyperparameters, lead to training pathologies. Others are related to the DNN design and structure that lead to performance degradation, whether the DNN is underfitting the data (i.e., low training and validation errors) or overfitting it (i.e., low training error and high validation error). The misconfiguration of a DNN training program impacts the effectiveness of the training process, and consequently, the quality of the trained DNN. However, they manifest themselves during the model learning process; so the debugging of the training program should help catch these undercover failures that are hard to detect at inference executions during the model testing.

Table 5.1 Real Bugs in DNN-based Software System

| **Category** | **Bug** | **Common Root Cause(s)** |
|---|---|---|
| **Coding Bug** | missing preprocessing | missing input normalization |
| | | missing output normalization |
| | wrong preprocessing | redundant data normalization |
| | wrong optimisation function | gradients with flipped sign |
| | missing softmax | missing softmax activation |
| | redundant softmax | softmax out-and in-the loss |
| | wrong type of activation | softmax for hidden activations |
| | | softmax for 1-dim output |
| | | over-restricted output domain |
| | wrong softmax implementation | softmax over wrong axis |
| | wrong loss function | CE over wrong axis |
| | | inverted CE's mean and sum |
| | | MSE with wrong broadcasting |
| | wrong data batching | shuffle only the features |
| | wrong data augmentation | invalid data transformation |
| **System Misconfi-guration** | wrong initialization | constant weights |
| | | dummy random weights |
| | wrong loss selection | use of MSE instead of CE |
| | suboptimal learning rate | a low learning rate |
| | | a high learning rate |
| | epsilon for optimiser too low | an Adam epsilon $\epsilon < 10^{-8}$ |
| | missing normalization layer | missing batch-norms |
| | | no-update of batch-norm globals |
| | missing regularisation | low $\lambda$ for norm penalties |
| | | high $\lambda$ for norm penalties |
| | | high $keep\_p$ for dropouts |
| | | low $keep\_p$ for dropouts |
| | unbalanced dataset | Labels are not equally distributed |

**Synthetic Tensorflow Buggy Programs**

Liu et al. [98] designed a base CNN that represents a typical CNN for image classification. Then, they derived diverse ineffective variants by poorly re-designing some parts of the CNN to evaluate the capacity of their visual diagnosis tool, CNNVis, in detecting the effects of the added poor design choices. Indeed, *TheDeepChecker* gauges different statistics and metrics on the DNN internals to detect fine-grained symptoms on the dysfunctioning or instability of DNN's components. The heuristic-based verification rules identify the occurred bug based on its effects on the training routines and flag the defective component relying on its current metrics' status. Therefore, our first evaluation of *TheDeepChecker* consists of an assessment on synthetic buggy DNN programs. Figure 5.7 shows our systematic approach, following the same methodology of Liu et al. [98], to create synthetic mutant DL programs, containing the above-mentioned DL faults. First, we select the Base DNN training programs for which the identified DL fault is applicable. Indeed, we prepare a mixed set of Base NNs in order to cover different architecture and activation functions that are related to a particular learning problem (i.e., either classification or regression), as well as, advanced techniques to regularize and stabilize the training of complex DL models (i.e., increasing the depth of the neural network enhances its complexity). However, adding arbitrarily hidden layers to have higher learning capacity would be unnecessary and may induce issues if it is applied on simple learning problems. Thus, we set up two base CNNs, ShallowCNN and DeepCNN, that have been proposed to solve two well-known classification problems with increasing complexity. Next, the injection of the DL bug in the Base program consists in mutating minimally the original source code based on its main root cause and the toolkit official documentation (i.e., Tensorflow). Then, we validate the presence of the DL bug-related symptoms; otherwise, further refinements should be performed. Generally, the symptoms observed for non-crashing bugs are any unexpected low model performance (i.e., accuracy or average error) or slow learning process. The bug reporters were able to perceive the symptoms based on their past DNNs training experiences or their comparisons with the original source (e.g., a research paper, or a tutorial). In a similar fashion, the reference performances of model and learning speed for our Base DL programs is already known and can be leveraged to check the success of the bug injection. Following the steps described in Figure 5.7, we are able to create a buggy synthetic DNN program for each matched pair of a base neural network and a single DL fault. This helps isolate the DL faults and assess the sensitivity and specificity of *TheDeepChecker* regarding each injected fault separately. At this level of controlled experiments, we could separate the valid fired checks that highlight the foreknown negative effects of the injected fault, and the false positives that point out to other irrelevant side effects (which could mislead the users during the debugging).

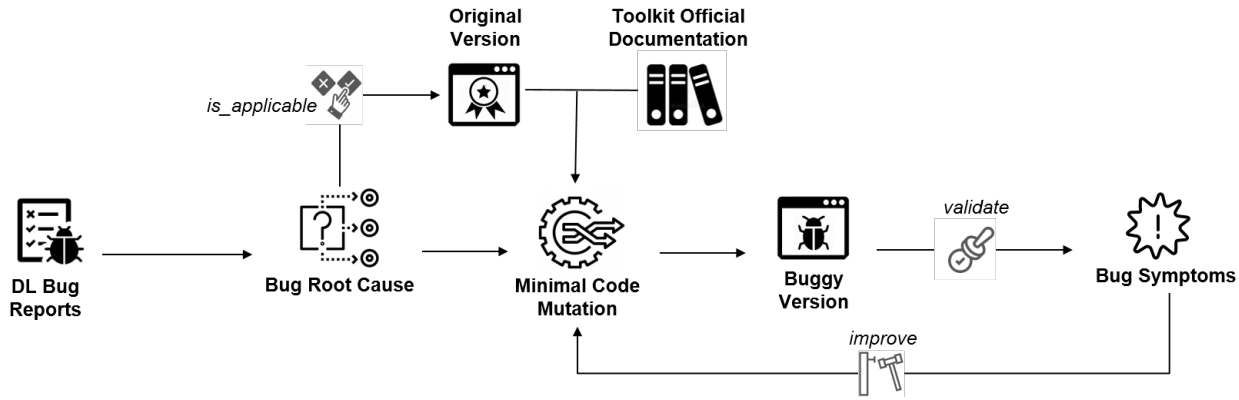



Figure 5.7 Overview of Synthetic DL Programs' Creation Steps

In the following, we describe the three Base NNs that we identified based on empirically-evaluated architecture, officially-debugged TF implementations, and publicly-available datasets.

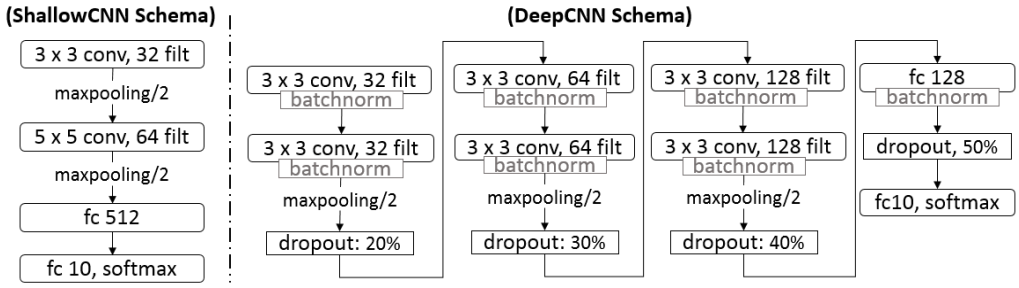



Figure 5.8 Schema of the two BaseCNNs Architecture

**RegrFNN.** The RegrFNN represents a basic feedforward neural network inspired from Google official examples [214], which contains two hidden fully-connected layers of 64-neurons with ReLU activation. It uses mean squared error (MSE) to compute the loss for regression problems. For regularization, we add both ridge regression (L2-norm of weights). Regarding the optimization, we use ADAM as a variant of gradient descent algorithm. It was trained on the classic Auto MPG Dataset [215] with the aim of predicting the fuel efficiency of late-1970s and early 1980s automobiles. The RegrNN reached less than 1.85 of mean absolute error on unscaled outputs within the range of $[10, 47]$.

**ShallowCNN.** The ShallowCNN is a LeNet [163] variant, containing two convolutional layers and two fully connected layers (more details in Figure 5.8). A max-pooling layer is set next to each convolutional layer. In addition, we select the widely used activation function, rectified linear unit (ReLU). As we target to solve a multi-class classification problem, we employed

softmax and cross-entropy to be respectively, the last output activation and the loss function. For regularization, we add both ridge regression (L2-norm of weights) and lasso regression (L1-norm of weights) as penalty terms to the loss function. Regarding the optimization, we use stochastic gradient descent (SGD), which remains the standard optimizer for training neural networks. It was trained on MNIST handwritten digits benchmark dataset [216], which includes $60,000$ labeled gray-scale images of size $28 \times 28$ in 10 labels. The dataset was split into a training set containing $50,000$ images and a test set containing $10,000$ images. The ShallowCNN achieved 99.35% accuracy rate on the test set.

**DeepCNN.** The DeepCNN is a VGG [151] variant, containing stacking three blocks of two convolutional layers followed by a max pooling layer and two fully-connected layers (more details in Figure 5.8). We also use ReLUs, softmax, and cross-entropy for activation, output and loss functions. However, we employ advanced regularization techniques to enable the training of this deep NN. We put a batchnorm layer next to each layer that stabilizes and accelerates the optimization process. In addition, each block or dense layer ends with a dropout layer using an increasing dropout rate (going from 20% to 50%) in order to offset the learning acceleration. Our optimization method is based on Adam, an advanced variant of gradient-based that can automatically adapt its learning rate within each optimized parameter. The DeepCNN was trained on CIFAR10 [217], which consists of $60,000$ labeled color images of size $32 \times 32$ in 10 different classes (e.g., airplane, bird, and truck). The dataset was split into a training set containing $50,000$ images and a test set containing $10,000$ images. The DeepCNN achieved 89.15% accuracy rate on the test set.

### Real TensorFlow Buggy Programs

Zhang et al. [138] reproduce several defective TF programs extracted from SO posts and GH projects. Among the categories of studied DL bugs, we found the Incorrect Model Parameter or Structure (IPS), which includes inappropriate modeling choices and algorithm configurations, degrading the quality of the training. Indeed, the major symptom of IPS bugs is low effectiveness, i.e., the performance of the on-training model does not improve as expected. After filtering out redundant TF programs and other incomplete models from their SO dataset, we ended up with 10 unique and full IPS buggy TF programs including data, model and training algorithm implementation. Table 5.2 presents the selected programs alongside the recommended fixes extracted from their related SO post discussions. Moreover, we clone 10 buggy versions of GH projects that correspond to bug-fixing commits in relation with IPS bugs. Table 5.3 shows the versions of GH projects with the implemented fixes that have been identified from the bug-fixing commit. Indeed, we emphasize that the fixes

with the buggy examples are added to mention the fault identified by the SO users or the GH project contributors, but there is no guarantee that it is the only bug or inefficiency in the project at that version. Technically, we follow the 'how to', libray's version, and official datasets, as described in the replication package of the empirical study [138] on Tensorflow Bugs, to prepare our set of buggy DL training programs.

Table 5.2 SO buggy TF-based training programs

| **Program** | **Recommended Fixes** |
|---|---|
| IPS-1 | switch to a numerical stable loss |
| IPS-4 | add mean to the loss |
| IPS-5 | change gradient descent by Adam |
| IPS-7 | add an output activation and a numerical stable loss |
| IPS-11 | remove the useless ReLU on the logits |
| IPS-12 | fix a typo in the accuracy function |
| IPS-13 | set lower learning rate($\eta$) and norm penalty($\lambda$) |
| IPS-14 | set a low learning rate($\eta$) |
| IPS-15 | set a low learning rate($\eta$) |
| IPS-17 | set a low learning rate($\eta$) |

Table 5.3 Github buggy TF-based training programs

| **Program** | **Recommended Fixes** |
|---|---|
| DLT_0edb182 | remove redundant softmax layers |
| DLT_20d1b59 | add a softmax layer |
| DLT_437c9c2 | improve the loss reduction |
| DLT_726b371 | add $\epsilon$ for a numerical stable loss |
| DLT_ded6612 | improve the parameters intialization |
| FCN-CTSCAN_b170a9b | fix a mistake in the loss function |
| TFE_333 | set an adequate weight initialization |
| TFE_368 | set a low learning rate($\eta$) |
| TFE_742675d | improve the loss reduction |
| TFE_bc09f95 | improve the loss reduction |

### Rule-based Debugger Baseline

As a baseline for automated training issue detection, we use SageMaker Debugger(*SMD*) [218], which is a fully managed debugging and monitoring service within the Amazon SageMaker platform for scalable machine learning in the cloud. *SMD* represents a framework-agnostic system to collect, query, and analyze data from ML model training and to automatically

capture issues using a set of built-in rules. Indeed, SageMaker automatically creates the training instance, pulls the training image from the Container Registry, and downloads data and training scripts into the container. Once the training is launched, *SMD* retrieves asynchronously the model data at specific intervals and uploads them to S3 bucket. Then, *SMD* runs the activated built-in or user custom rules in independant processing jobs on separate containers in a way that they do not interfere with the training job. Finally, users can set up alarms within Amazon Cloudwatch, which is a real time monitoring and observability service, to indicate when a rule is triggered. Table 5.4 summarizes the *SMD*'s built-in rules that are applicable for feedforward neural networks.

Table 5.4 SMD Built-in Rules Applicable to FNNs

| **Component** | **Rules** | **#id** |
|---|---|---|
| Data | Class Imbalance | $R_0$ |
| | Non-normalized image input | $R_1$ |
| Loss | Not Decreasing | $R_2$ |
| | Unchanged | $R_3$ |
| | Overfitting | $R_4$ |
| | Underfitting | $R_5$ |
| | Overtraining | $R_6$ |
| Weights | Poor Initialization | $R_7$ |
| | Abnormal Update Ratio | $R_8$ |
| | All Zeros | $R_9$ |
| | Abnormal Variance | $R_{10}$ |
| | Exploding | $R_{11}$ |
| Activations | Neurons Saturation | $R_{12}$ |
| | Dying Neurons | $R_{13}$ |
| Gradients | Vanishing | $R_{14}$ |
| | Exploding | $R_{15}$ |

Although *SMD* and *TheDeepChecker* target a common subset of training issues, the logic of debuggers' rules are quite different, as *SMD* relies exclusively on the model data collected offline during the user-configured training session. However, *TheDeepChecker* is an online debugger framework that takes control of the ML training program and the datasets to orchestrate the running of a sequential multi-phase debugging workflow, including preliminary independent checks, a monitored single-batch training, and comparisons of multiple trained models. Indeed, the *TheDeepChecker*'s debugging workflow involves verification rules that validate necessary properties of the training program and assess the status of heuristic-based detectors for common DL model training issues.

### 5.3.2 Case Study Results

In the following, we present the results of debugging sessions using *TheDeepChecker* on both synthetic and real-world buggy programs. First, we evaluate the capacity of *TheDeepChecker* in detecting faults injected in the base NNs, whether the fault is a coding bug or a misconfiguration of the system. Then, we assess the effectiveness of *TheDeepChecker* in debugging real-world buggy TF programs that may contain multiple hidden bugs.

**Debugging DNN Software System that contains Coding Bugs.**

Table 5.5 shows the results obtained on the synthetic buggy training programs. For each injected bug, we report all the checks fired by *TheDeepChecker*, as well as the performance of the trained model on test data and the *SMD*'s rules that detected issues.

**Accuracy of *TheDeepChecker* on coding bug detection.** Given the fault injected in the synthetic example, we put in bold the fired check(s) that are considered to be conceptually connected to the underlying issue. Indeed, these bolded checks could guide the user towards recognizing the occurred issue and fixing the buggy training program. Moreover, we study the other fired checks for a full assessment as we keep running the *TheDeepChecker*, but we do not need to wait for all the fired checks to spot and fix alerted DL bugs. Thereby, we also provide a user-defined boolean setting (*failed_on=True/False*) that would enable an exception to be raised whenever *TheDeepChecker* encounters any of these verifications. In practice, this helps save useless running time and computational resources. Thus, we calculate the number of True Positive (TP) checks, False Positive (FP) checks and False Negative (FN) checks. True positives are represented as $a+b$, where $a$ and $b$ are the number of checks, respectively, defined to catch precisely the injected bug and to identify generic training difficulties that can be correlated to various DNN issues. Although the proposed generic checks, counted in $b$, spot fine-grained inefficient training traits, we do not consider the bug was detected in case of $a = 0$. Therefore, *TheDeepChecker* has a precision of 90% and a recall of 96.4% in detecting the synthetized DL coding bugs.

**Comparison with Baseline.** As can be seen, missing code statements can result in the dysfunction of a component in the DNN training program. In such cases, our verification routines find some violations of properties associated with the expected output and normal behavior of the buggy component. First, missing or redundant input normalization and overscaled outputs are detected by *TheDeepChecker* using the following prechecks on the loaded data: unscaled inputs and unscaled outputs. Next, unapplying softmax over outputs and unshuffling the labels would trigger the following verification routines, respectively, invalid predictions (i.e., do not respect the probability laws) and corrupted labels (i.e., labels do not

match with the features). Another studied data preprocessing bug is the inappropriate data transformers that induce a shift between original and augmented datasets. *TheDeepChecker* was able to detect such data shifts from the perspective of the trained model behavior by comparing the activation patterns of the neural networks trained on augmented/original datasets. In relation to the data, *SMD* supports only a rule ($R_1$) for unnormalized images detection. For the missing activation, *SMD* triggers vanishing gradients ($R_{14}$), which is a quite generic issue for DNN training.

Concerning wrong coding statements, *TheDeepChecker* is still accurate for *mistaken axis in tensor-based operations*, thanks to the gradient-based dependency verification that point out any overlapping between instances (i.e., rows) caused by an incorrect calculation. However, the dependency verification fails on the reproduced wrong broadcasting of MSE because the error does not induce an overlap between instances. In most of these wrong coding statements, *SMD* triggers rules indicating the difficulty of training such as vanishing gradients ($R_{14}$) and non-decreasing loss ($R_{14}$). Both *TheDeepChecker* and *SMD* succeed in revealing coding faults that change the training algorithm's behavior, like flipped signs in the gradient calculation that turns the training process into a loss maximization. Nevertheless, *TheDeepChecker* is capable of detecting a wrong cross-entropy function with switched mean and sum operations based on the initial loss value which is larger than expected, and consequently, indicate the presence of a wrong reduction function for the losses over data inputs. *SMD* reports vanishing gradients ($R_{14}$) for only the ShallowNN that could not train the model with a wrong loss reduction (i.e., the accuracy of the trained model is equivalent to random guess).

Regarding the misuse of API functions, we reproduced the situation of redundant softmax in-and out-the cross-entropy loss for both studied Base CNNs. Despite the fact that *TheDeepChecker* does not contain a specialized check to pinpoint accurately the issue, it could successfully detect its presence in the training program for both buggy examples, contrary to *SMD* that remains silent. In fact, *TheDeepChecker* generated an error message reporting learning issues in the last dense layer of both ShallowCNN and DeepCNN containing the softmax redundancy within the loss; more specifically, a slowness learning issue due to either low updates' ratio. These slowness learning issues, when spotted at the last layer (i.e., softmax activation), strongly indicate a waste of information and an obstruction of the back-propagation of the error through the two consecutive softmax activations. Hence, the relation between the fired checks and the occurred bug is not straightforward (i.e., we do not count it among the main fired checks (i.e., the number $a$ in True Positives). Nevertheless, it is worth mentioning that *TheDeepChecker* generated a warning for the user about the slow weight update encountered exclusively in the last dense layer, which can lead him to

review this ending part of the DNN design which contains the issue (i.e., redundant/useless last activation function). Besides, *TheDeepChecker* reported symptoms in relation to difficulties faced in the loss minimization: slow decreasing loss for both subjects and additionally highly-fluctuating loss for DeepCNN.

Overall, *TheDeepChecker* reported misleading checks which are considered as false alarms. First, it mistakenly reports non-representative loss (i.e., loss measure is not correlated with performance metric) in some DeepCNN cases. Indeed, the obtained highly-fluctuating loss caused by the activation redundancy significantly reduced the magnitude of the correlation between the resulting noisy loss and the classification accuracy, and triggered the verification routine related to non-representative loss. This highlights the difference between shallow and deep CNNs and how the loss landscape is more complex for deeper NNs and can be substantially affected by these relatively minor changes in the math calculations involved in the DNN mapping function. Second, the traditional regularization check consistently triggers an alert of overwhelming regularization gradient cost over the data gradient loss, but our inspection leads us to the fact that actual issue was the vanishing gradient problem reaching exactly 0; we consider as misleading because the check was designed to capture over-regularization cost and it can mislead the user towards inspecting unnecessarily the regularization. On the other hand, we unbolded the *SMD*'s rules that have been fired even for the clean training program. Indeed, these rules, $R_8$ and $R_{10}$, were fired during the early training iterations on the DeepNN that includes advanced regularizers to smooth the loss landscape and be able to train several hidden layers. This can be explained by the high sensitivity of these rules that alert quickly about abnormal weights' update ratio and variance starting from the warm-up period required by these regularizers to stabilize the training. Besides, *SMD* reports a false alarm of poor weight initialization ($R_7$) on a wrong cross-entropy calculation. In fact, *SMD* relies on the application of rules' functions on the periodically-saved summary statistics. Hence, $R_7$ checks the variance of activation inputs across layers, the distribution of gradients, and the loss convergence for the initial steps to determine if a neural network has been poorly initialized. These training inefficiencies can be the result of many issues in the training program, which increases the risk of misleading alerts.

Table 5.5 Results of debugging coding bugs in DNN-based software systems

| Faults | Base NN | Perf. | SMD Rule(s) | Fired Check(s) | TP | FP | FN |
|---|---|---|---|---|---|---|---|
| **missing input normalization** | Regr | 24.20 | - | **Uns-Inps**[1], PI-Loss[2] Un-Fit-Batch[3], Uns-Act-HS[4] | 1+3 | 0 | 0 |
| | Shallow | 11.35% | $R_1$,$R_8$,$R_{14}$ | **Uns-Inps**, PI-Loss, Un-Fit-Batch Div-Loss[5], Div-W[6], Div-B[7], Div-Grad[8] | 1+6 | 0 | 0 |
| | Deep | 85% | $R_1$,$R_8$,$R_{10}$ | **Uns-Inps**, PI-Loss, Uns-Act-HS, NR-Loss[9] | 1+2 | 1 | 0 |
| **over-scaled outputs** | Regr | 20.14 | $R_2$, $R_{12}$ | **Uns-Outs**[10], SD-Loss[11], Dead-ReLU[12], Uns-Act-HS | 1+3 | 0 | 0 |
| **redundant input normalization** | Regr | 2.86 | - | **Uns-Inps**, SD-Loss, Uns-Act-LS[13], Un-Fit-Batch | 1+3 | 0 | 0 |
| | Shallow | 33.75% | $R_8$, $R_{14}$ | **Uns-Inps**, SD-Loss, W-Up-Slow[14], Uns-Act-LS | 1+3 | 0 | 0 |
| | Deep | 77.5% | -,$R_8$,$R_{10}$ | **Uns-Inps**, Uns-Act-LS | 1+1 | 0 | 0 |
| **gradients with flipped sign** | Regr | 1.72e7 | - | Un-Fit-Batch, **Div-Loss**, Uns-Act-HS | 1+2 | 0 | 0 |
| | Shallow | 9.8% | $R_{11}$,$R_{14}$ | Un-Fit-Batch, **Div-Loss**, Div-W, Div-B, Uns-Act-HS, Van-Grad[15] | 1+5 | 0 | 0 |
| | Deep | 10% | $R_{11}$,$R_{14}$ | Un-Fit-Batch, **Div-Loss**, Uns-Act-HS, NR-Loss[16] | 1+2 | 0 | 0 |
| **missing softmax activation** | Shallow | 9.8% | $R_{14}$ | PI-Loss, **Inv-Outs**[17], SD-Loss W-Up-Slow, Van-Grad, Un-Fit-Batch, Over-Reg-Loss[18] | 1+5 | 1 | 0 |
| | Deep | 11.48% | $R_{14}$,$R_8$,$R_{10}$ | PI-Loss, **Inv-Outs**, Van-Grad | 1+2 | 0 | 0 |
| **softmax out-and in-the loss** | Shallow | 99.29% | - | SD-Loss, W-Up-Slow(Dense) | 0+2 | 0 | 1 |
| | Deep | 83.24% | -,$R_8$,$R_{10}$ | SD-Loss, HF-Loss[19], W-Up-Slow(Dense), NR-Loss[20] | 0+2 | 1 | 1 |
| **softmax over wrong axis** | Shallow | 99.45% | $R_{14}$ | PI-Loss, **Inv-Outs**, **Inv-Out-Dep**[21], Inv-Loss-Dep[22] | 2+2 | 0 | 0 |
| | Deep | 85.86% | $R_{14}$,$R_8$,$R_{10}$ | PI-Loss, **Inv-Outs**, **Inv-Out-Dep**, Inv-Loss-Dep | 2+2 | 0 | 0 |
| **CE over wrong axis** | Shallow | 8.92% | $R_2$,$R_7$ | **PI-Loss**, **Inv-Loss-Dep** | 2+0 | 0 | 0 |
| | Deep | 86.79% | -,$R_8$,$R_{10}$ | **PI-Loss**, **Inv-Loss-Dep** | 2+0 | 0 | 0 |
| **MSE with wrong broadcasting** | Regr | 7.02 | $R_2$ | Un-Fit-Batch, SD-Loss, Van-Grad | 0+3 | 0 | 1 |
| **inverted CE's mean and sum** | Shallow | 11.34% | $R_{14}$ | **PI-Loss** | 1+0 | 0 | 0 |
| | Deep | 87.08% | -,$R_8$,$R_{10}$ | **PI-Loss** | 1+0 | 0 | 0 |
| **shuffle only the features** | Regr | 7.27 | - | **Corrupted Labels** | 1+0 | 0 | 0 |
| | Shallow | 11.35% | - | **Corrupted Labels** | 1+0 | 0 | 0 |
| | Deep | 10.09% | -,$R_8$,$R_{10}$ | **Corrupted Labels** | 1+0 | 0 | 0 |
| **invalid data transformation** | Shallow | 99.24% | - | **Shifted-Augmented-Data** | 1+0 | 0 | 0 |
| | Deep | 86.28% | -,$R_8$,$R_{10}$ | **Shifted-Augmented-Data** | 1+0 | 0 | 0 |

## Debugging Misconfigured DNN Software System.

Table 5.6 presents the debugging results of *TheDeepChecker* on misconfigured synthetic training programs, following a similar structure as Table 5.5.

[1] Unscaled Inputs [2] Poor Initial Loss [3] Unable to fit Single Batch [4] Unstable Activation with High STD [5] Diverging Loss [6] Diverging Weights [7] Diverging Biases [8] Diverging Gradient [9] Non-Representative Loss [10] Unscaled Outputs [11] Slow-Decreasing Loss [12] Dead ReLU [13] Unstable Activation with Low STD [14] Weight Update Slowly [15] Vanishing Gradients [16] Invalid Outputs [17] Overwhelming Regularization Loss [18] High-Fluctuating Loss [19] Invalid Output Dependency [20] Invalid Loss Dependency

**Accuracy of *TheDeepChecker* on misconfigurations detection.** We calculate true positive, fault positive, and fault negative rates alongside the list of fired checks. Overall, *TheDeepChecker* achieved 77% precision and 83.3% recall, when detecting misconfigurations in the studied DNN training programs.

**Comparison with Baseline.** As can be seen, inappropriate initial weights, i.e., constant and inept randomness, correctly trigger the following *TheDeepChecker*'s checks, *unbreaking symmetry* and *poor weights initialization*. *SMD* uses multiple indirect criteria to recognize a bad initialization through activations' variances, gradients' distributions, and the loss curve. It spots the unbreaking symmetry issue, but it was less effective in detecting the dummy random weights with inappropriate variance.

Similarly, *TheDeepChecker* was able to detect missing stabilization components, including missing the whole batchnorm layers and missing the update of their global statistics, through respectively, high unstable activations and unstable transfer from train to inference mode. Also, weak regularization triggers the checking rule of zero loss, which implies that the optimizer likely overfits the given training batches and strong regularization can lead to overwhelming weight norm penalty (in case of l2-norm) or unstable transfer from train to test mode (dropout). Nonetheless, *SMD* was only capable of detecting the negative effects of strong l2-norm regularization on the weights' variances and dying ReLUs.

Moreover, architecture- or problem-dependant issues like using ineffective loss function are challenging to detect for developers (e.g., we refer to SO posts #36515202, #49322197, #56013688, and #62592858); their identification depends heavily on the knowledge of the developer on Deep Learning and his experience in implementing DNN programs. Indeed, these issues can have severe effects on the convergence and stability of the DNN training process, especially on relatively high capacity DNNs and complex statistical learning problems. When it comes to regression problems, the use of cross-entropy(CE) instead of mean squared error(MSE) hinders the learning; so both approaches were able to detect that. In case of classification problems, the use of MSE instead of CE had less effects on the training performance, and consequently, harder to automatically detect. Thus, *TheDeepChecker* alerted only inefficient training traits including vanishing gradients and slow updating parameters for the deepCNN. Inversely, *SMD* alerted abnormal training curve, including non decreasing and unchanged loss for the ShallowCNN, but the resulting test performance of the model is high enough to just conclude that this steadiness was due to the optimization convergence.

With less implicit issue-verification connections, poor choices regarding the optimization routines, including inadequate magnitude of the learning rate or the epsilon of null divider prevention, influence negatively the speed of parameters learning. Only *TheDeepChecker*

successfully detected the substantial difference in the magnitude of parameters' updates. It detected slow updating parameters in case of low learning rate and fast updating parameters in case of high learning rate and low Adam epsilon. Indeed, *TheDeepChecker*'s unstable weight update detection strategy reposes on DL researchers' recommendations and it is sensitive enough to report precisely the low or high magnitudes of weight updates that can guide DL users to adjust the optimizer's configuration touching the extent of updates. Nevertheless, both compared debugging methods were able to the divergence of ShallowCNN caused by the high learning rate that manifests through a training stagnation. Indeed, the high learning rate quickly induces a large update step towards the gradient direction, which takes the weights to nonoptimal regions, and consequently, provokes dead ReLUs and a highly-fluctuating loss optimization without convergence (unable to overfit the batch). Since the inadequate optimizer's jump likely happens at the very first iterations, there would not be fired checks in relation with the speed of learning; so the users cannot identify the root cause easily based on all this checking report.

As above-discussed for DL coding bugs 5.3.2, *SMD* keeps consistently reporting both of $R_8$ and $R_{10}$ rules, as well as, it reports a false alarm of poor weight initialization ($R_7$) on a inappropriate loss selection (CE instead of MSE). On the other hand, *TheDeepChecker* reports false positives of non-representative loss, overwhelming regularization loss paired with, respectively, high fluctuating loss and vanishing gradients. This reinforces the fact that these false positives are caused by the non-consideration of potential dependencies between DL training issues.

In the future, we plan to analyze in-depth the sequence of fired checks for the different DL bug types and extend *TheDeepChecker* to consider recurrent patterns of negative verifications in order to not only reduce the number of false positives that we discover during the assessment, but also avoid overwhelming the users by several correlated training issues. Indeed, we observe in both results (i.e., Tables 5.6 and 5.5) repetitive co-occurrences of multiple checks. For instance, we see that various errors make the weight updates unstable, so the weights can potentially have more and more negative values (i.e., over-negative tendency). In this case, the layers' weighted sum would produce mainly negative quantities, which will turn into zero activations by the ReLUs (i.e., dead). Then, the back-propagated gradients, as described in Equation 5.1, starts vanishing quickly and, as a result, the weights' updates will tend to have too low magnitude and could cause the DNN to freeze (i.e., triggering likely a slow- or non-decreasing loss).

Table 5.6 Results of debugging misconfigurations in DNN-based software systems

| Faults | Base NN | Perf. | SMD Rule(s) | Fired Check(s) | TP | FP | FN |
|---|---|---|---|---|---|---|---|
| **constant weights** | Regr | 2.53 | $R_7$,$R_{10}$ | **Un-Sym-W**[23], SD-Loss, Uns-Act-LS W-Up-Slow, Un-Fit-Batch | 1+4 | 0 | 0 |
| | Shallow | 11.35% | $R_7$,$R_{10}$,$R_{13}$,$R_{14}$ | **Un-Sym-W**, SD-Loss, Neg-W[24], Over-Reg-Loss, Uns-Act-LS, Dead-ReLU, W-Up-Slow, Van-Grad, Un-Fit-Batch | 1+8 | 0 | 0 |
| | Deep | 75.92% | $R_7$,$R_{14}$,$R_8$,$R_{10}$ | **Un-Sym-W**, SD-Loss, Uns-Act-HS, W-Up-Slow | 1+3 | 0 | 0 |
| **dummy random weights** | Regr | 2.17 | - | **PI-W**[25], Uns-Act-LS | 1+1 | 0 | 0 |
| | Shallow | 99.18% | $R_2$,$R_7$ | **PI-W**, PI-Loss, SD-Loss Dead-ReLU, Uns-Act-LS | 1+4 | 0 | 0 |
| | Deep | 71.89% | -,$R_8$,$R_{10}$ | **PI-W**, Uns-Act-HS, NR-Loss | 1+1 | 1 | 0 |
| **use of MSE instead of CE** | Shallow | 99.17% | $R_2$,$R_3$ | - | 0 | 0 | 1 |
| | Deep | 69.52% | -,$R_8$,$R_{10}$ | SD-Loss, HF-Loss, W-Up-Slow | 0+3 | 0 | 1 |
| **use of CE instead of MSE** | Regr | 49.46 | $R_3$,$R_7$,$R_1$4 | Un-Fit-Batch, Van-Grad(dense), Over-Reg-Loss | 0+3 | 1 | 1 |
| **low learning rate** | Regr | 5.48 | - | Un-Fit-Batch, SD-Loss, **W-Up-Slow** | 1+2 | 0 | 0 |
| | Shallow | 98.96% | - | SD-Loss, **W-Up-Slow** | 1+1 | 0 | 0 |
| | Deep | 53.73% | -,$R_8$,$R_{10}$ | SD-Loss, **W-Up-Slow**, NR-Loss | 1+1 | 1 | 0 |
| **high learning rate** | Regr | 2.55 | - | **W-Up-Fast**[26], SD-Loss | 1+0 | 0 | 0 |
| | Shallow | 11.34% | $R_1$3,$R_1$4 | Un-Fit-Batch, HF-Loss, Dead-ReLU, Uns-Act-LS, NR-Loss | 0+4 | 1 | 1 |
| | Deep | 86.29% | -,$R_8$,$R_{10}$ | Uns-Act-HS, **W-Up-Fast** | 1+1 | 0 | 0 |
| **Adam epsilon** $\epsilon < 10^{-8}$ | Deep | 86.75% | -,$R_8$,$R_{10}$ | Uns-Act-HS, **W-Up-Fast** | 1+1 | 0 | 0 |
| **missing batch-norms** | Deep | 80.79% | - | SD-Loss, **Uns-Act-LS**, W-Up-Slow, NR-Loss | 1+2 | 1 | 0 |
| **no-update of batch-norm globals** | Deep | 84.35% | -,$R_8$,$R_{10}$ | **Uns-Mode-Tr** | 1+0 | 0 | 0 |
| **low** $\lambda$ **for norm penalties** | Regr | 2.39 | - | **Zero-Loss** | 1+0 | 0 | 0 |
| **low** $\lambda$ **for norm penalties** | Shallow | 99.27% | - | **Zero-Loss** | 1+0 | 0 | 0 |
| **high** $\lambda$ **for norm penalties** | Regr | 7.05 | $R_1$0,$R_1$3 | **Over-Reg-Loss**, Uns-Act-LS, Un-Fit-Batch | 1+2 | 0 | 0 |
| **high** $\lambda$ **for norm penalties** | Shallow | 64.88% | $R_1$0,$R_1$3 | **Over-Reg-Loss** | 1+0 | 0 | 0 |
| **high** *keep_p* **for dropouts** | Deep | 73.15% | - | **Zero-Loss** | 1+0 | 0 | 0 |
| **low** *keep_p* **for dropouts** | Deep | 78.74% | -,$R_8$,$R_{10}$ | HF-Loss, **Uns-Mode-Tr**,NR-Loss | 1+1 | 1 | 0 |
| **unbalanced dataset** | Shallow | 99.24% | bm$R_0$ | **Unbalanced Labels** | 1+0 | 0 | 0 |
| | Deep | 86.28% | bm$R_0$,$R_8$,$R_{10}$ | **Unbalanced Labels** | 1+0 | 0 | 0 |

### Debugging Runtime Evaluation

*TheDeepChecker* enables the debugging of DNN training programs through performing a stack of checks prior, during, and after the program execution. Executing these checks can be expensive.

In this section, we assess the execution cost of the checks on a DNN training program. The execution was done using a single machine having a CPU Intel i7-8750H 6-cores and a GPU NVIDIA GeForce RTX 2080. The evaluation was done using the above-described base neural networks. To provide insights on the execution cost of *TheDeepChecker* during a debugging session, Table 5.7 reports the average time spent using *TheDeepChecker* on each phase. As can be seen, the pre- and post-training phases, (1) pre-training conditions check and (3) post-fitting conditions check, varies according to the dimensionality of the input data and the complexity of NN as they include normalization tests, tensor-based operations dependency, and even regular training of DNN for several epochs (we set up 50 by default) to compare metrics over epochs and make some activation patterns comparison. Next, the

[1] Unbreaking Symmetry Weight [2] Poor Initial Weight [3] Weight Update Quickly [4] Unstable Transfer Mode

phase of proper fitting on a single batch requires only few iterations (i.e., it remains a setting option for running the test and by default, a maximum iterations equals to 200), but with short periodicity of verification routines (i.e., it is also a setting option and by default, we fixed a period equals to 10 iterations). These default setting options are derived from our experimentation, however, the configuration choices should take into consideration the complexity of the DNN under test and the default settings enable a sufficient amount of monitored iterations to detect the issues given the complexity of the studied neural networks. For higher complexity NNs, an increase of these parameters' values may enhance the issue detection capability of *TheDeepChecker* as it will make more intensive verification during the testing iterations. Given the average execution time of a regular training iteration and a monitored training iteration, we find that the verification routines running in-between the training iterations increased the training iteration runtime by averagely 10×. In contrast, *SMD* hooks multiple internal data recorders to the full training session in order to fetch and save periodically tensors including activations, weights, and gradients. *SMD* incorporates several optimizations to improve I/O performance and sets up relatively long, by default, save intervals (i.e., around 500 steps). In fact, our experimentation on AWS instance, **ml.p2.xlarge**, which contains 1-GPU and 4 virtual processors, yields, averagely, an overhead of no more than 13% for a training iteration monitored by 4 rules. Then, *SMD* verifies the rules by offloading data inspection, shared into separate containers in a way that users can run an arbitrary number of rules without impacting the training process itself. Nevertheless, this multi-job processing and I/O data offloading adds an overhead, even for simple DNN programs. During our debugging sessions on **ml.p2.xlarge** with 4 activated rules, we wait for $3-5$ minutes to have the final rules check reports.

Table 5.7 Execution cost of *TheDeepChecker* during different debugging phases (average time in seconds)

| | **Pre-train Check** | **Fitting Check** | **Post-Fit Check** |
|---|---|---|---|
| **ShallowCNN** | 16.63 | 20.61 | 751.70 |
| **DeepCNN** | 20.81 | 82.74 | 1641.59 |
| **RegrCNN** | 6.75 | 0.996 | 5.70 |

**Assessment of *TheDeepChecker* on Real Buggy DNN Software**

Table 5.8 and Table 5.9 show the debugging results of both *TheDeepChecker* and *SMD* on real-world buggy TF programs extracted, respectively, from Stackoverflow and GitHub. Indeed, we add all the turned-on verification checks/rules on each tested DNN program, but we highlight the ones that are considered to be related to the actual fixed bug. For *SMD*, we rely

on its official built-in rules documentation [219] about their logic and targeted issues in order to decide if the fired rules have This allows us to compute the success rate of *TheDeepChecker* in detecting the bugs that have been fixed by the SO users or the GH project maintainers. Indeed, *TheDeepChecker* succeeds in 70% of the SO buggy code and 80% of the buggy versions of GH projects. However, *SMD* succeeds in 60% of both SO and GH buggy examples. *SMD* generally alerts the user with high-level indicators of abnormal/suspicious on-training neural network state, but *TheDeepChecker* often reports broken properties that are connected to a narrower scope of DL faults and help users identify the main root cause. For instance, lines of code 5.1 shows that the NN's output layer, *y_*, has no activation function, which causes an incorrect calculation of cross entropy using logits instead of probabilities. Moreover, the cross entropy formula involves a risky use of log on possibly zero values.

```
y_ = tf.matmul(h1,W_out)
cross_entropy = tf.reduce_sum(-y*tf.log(y_)-(1-y)*tf.log(1-y_),1)
```

Pseudo-Code 5.1 Lines 18;21 of IPS-7

*TheDeepChecker* reports invalid output layer because the verification routine on the last layer requires yielding probabilities when NN solves a classification problem. It also alerts about the diverging loss caused by the NaNs of *tf.log(0)*. Although abnormal variance of weights reported by *SMD* reflects an unstable learning process, it cannot be connected to the missing activation or the unstable loss issues.

```
Yhat = tf.matmul(l3, W5) + b5
Ypred = tf.nn.sigmoid(Yhat)
# ...
correct_prediction = tf.equal(tf.greater(Y, 0.5), tf.greater(Yhat, 0.5))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Pseudo-Code 5.2 Lines 37-38;45-46 of IPS-12

Code 5.2 shows another buggy code snippet, where *SMD* reports only abnormal variance of weights, contrary to *TheDeepChecker* which spots the non-correlation anomaly between the loss and the accuracy. Indeed, the user mistakenly used the logits *Yhat* instead of the sigmoid outputs *Ypred* in the inference of predictions with a threshold of 0.5, which leads to a wrong calculation of accuracy. Only *TheDeepChecker* cover the coding mistakes in the performance functions through the validation of their correlation coefficient over the training iterations. Thus, the use of the logits instead of probabilities would always yield the class 1 for predictions; which would break the property of performance metrics correlation as the

progress of accuracy metric would be uncorrelated with the loss value.

```
def fc_layer(input, size_in, size_out, name="fc"):
    with tf.name_scope(name):
        w = tf.Variable(tf.truncated_normal([size_in, size_out], stddev=0.1))
        b = tf.Variable(tf.constant(0.1, shape=[size_out]))
        activation = tf.nn.relu(tf.matmul(input, w) + b)
        # ...
        return activation

def mnist_model(learning_rate, path):
    # ...
    logits = fc_layer(fc1,1024,10,name="fc2")
    probabilities = tf.nn.softmax(logits)

    with tf.name_scope("xent"):
        xent = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(logits=logits,labels=y))
```

Pseudo-Code 5.3 Lines 20-28;46-51 of IPS-11

Even if it has no specific property that would be broken by the fault, *TheDeepChecker* can trigger training difficulty symptoms equivalent to *SMD*'s rules, as shown in the lines of Code 5.3. Indeed, the DL developer unified all the fully-connected layers by a custom function, but he mistakenly applied it over the last fully-connected layer. This induces a useless ReLU activation on the logits before computing softmax-based scores or losses. As a result, the double activation obstructs the information from flowing smoothly and causes a slowness of weight update and vanishing of gradients. Although the same fault-related symptoms were reported by both debugging tools, *TheDeepChecker* reports the *unable to overfit the batch* that strongly indicate a serious model fitting problem.

Nonetheless, *TheDeepChecker* was more verbose triggering several positive checks for each buggy DL training program, and relying on the accepted answer on SO post or the changes in the bug-fixing commit does not allow us to compute the false positives, which represent fired checks without corresponding issue, and false negatives, which represent the existed faults without corresponding fired checks, on these real-world DNN programs. Moreover, finding fired verification routines that are, by definition, related to the actual fixed bug, does not imply that users can fix the bug correctly using the *TheDeepChecker* diagnostic reports. In the next evaluation of usability, we will ask two experienced DL engineers to perform

a closed-feedback debugging and fixing loop using *TheDeepChecker* on each of the buggy snippets of code found on StackOverflow.

### 5.3.3 Usability of *TheDeepChecker*

In this section, we report about a usability study performed with two professional DL engineers, namely $E_1$ and $E_2$, with the aim to assess the relevance of *TheDeepChecker*'s error messages at guiding developers in identifying the root cause of bugs and fixing them. The two DL engineers involved in the study have 3 years of experience working with TensorFlow. They are currently employed in AI software development teams in technology companies, building DL-based software systems. To assess the relevance of *TheDeepChecker*'s error messages, we provided the two engineers with 10 buggy DL training programs (i.e., the ones reproduced from SO posts in Table 5.8) and asked them to use *TheDeepChecker* for debugging them and fixing the identified bugs. We made it clear to them that we are not evaluating their ability to detect the bugs based on their knowledge and asked them to only follow the clues contained in the debugging logs generated by *TheDeepChecker*. We also asked them to explain how they inferred the root cause of faults based on the information provided by *TheDeepChecker* and to suggest fixes. We made the decision to ask participants to fix detected bugs because to allow them to explore as many bugs as possible. The DL programs used in our study contain more than one bug, and to progress in the debugging process participants have to fix detected bugs. For instance, unnormalized inputs may cause the divergence of the training and turn the loss quickly to NaN. Hence, it obstructs the training dynamics, and consequently, the *TheDeepChecker*'s debugging session. Thus, the DL engineer should normalize the inputs to fix the issue and restart the debugging session. In Table 5.8, we added $(n)$ next to verification routines to identify the debugging session during which the routine was triggered. For example, Zero-Loss(2) means that during the second debugging session (2) after fixing some of the issues, *TheDeepChecker* newly reports the Zero-Loss warning that indicates strongly a lack of regularization in the DNN.

Although the bug fixes suggested by the two engineers are quite different, they have performed similar sequence of debugging sessions where they focus on fixing the same training issues at each session and have received mostly the same amount of notification messages (from the fired verification routines) at the different debugging steps for each given buggy program.

Table 5.10 shows the fixes suggested by the engineers based on the error messages generated

[1] Poor initial bias [2] Missing bias [3] Saturated Sigmoid

Table 5.8 Debugging Results of StacOverflow TF-based training programs

| **Program** | *TheDeepChecker*'s **Fired Check(s)** | **SMD Rule(s)** |
|---|---|---|
| IPS-1 | PI-W, PI-b, PI-Loss, Uns-Act-LS | $R_7$, $R_{10}$ |
| IPS-4 | PI-W(1), PI-b(1), **PI-Loss**(1), Uns-Act-HS(1), Zero-Loss(2) | $R_2$, $R_{14}$ |
| IPS-5 | Uns-Inps, **Un-Fit-Batch**, **Div-Loss**, **W-Up-Fast**, W-neg | $R_9$, $R_{10}$ |
| IPS-7 | PI-W(1), Miss-b(1), PI-Loss(1), **Inv-Outs**(1), **Div-Loss**(1), Un-Fit-Batch(1), Sat-Sigmoid(1-2) | $R_{10}$ |
| IPS-11 | PI-W(1), PI-b(1), PI-Loss(1), **Van-Grad**(1), **W-Up-Slow**(1), **Un-Fit-Batch**(1), Zero-Loss(2) | $R_8$, $R_{10}$, $R_{14}$ |
| IPS-12 | Uns-Inps(1), PI-W(1), PI-Loss(1), **NR-Loss**(1), HF-Loss(2), Dead-ReLU(2), Uns-Mode-Tr(2) | $R_9$, $R_{10}$ |
| IPS-13 | Uns-Inps(1), PI-W(1), PI-Loss(1), **W-Up-Fast**(2), **Over-Reg-Loss**(2), **Un-Fit-Batch**(1-2) | - |
| IPS-14 | PI-W, PI-Loss, Un-Fit-Batch | $R_8$, $R_9$, $R_{10}$ |
| IPS-15 | Uns-Inps, Miss-b, **Div-Loss**, **Div-W**, **Div-Grad**, Un-Fit-Batch | $R_9$, $R_{10}$ |
| IPS-17 | Uns-Inps(1), Div-Loss(1), Un-Fit-Batch(1), W-Up-Slow(2) | $R_9$, $R_{10}$ |

Table 5.9 Debugging Results of Github TF-based training programs

| **Program** | *TheDeepChecker*'s **Fired Check(s)** | **SMD's Fired Rule(s)** |
|---|---|---|
| DLT_0edb182 | PI-W, PI-B, Uns-Act-LS, SD-Loss | $R_{14}$ |
| DLT_20d1b59 | PI-W, PI-B, PI-Loss, Uns-Act-LS, SD-Loss, NR-Loss, Un-Fit-Batch | $R_{14}$ |
| DLT_437c9c2 | Un-Sym-W, **PI-Loss(Huge Err)** | $R_8$, $R_9$, $R_{10}$ |
| DLT_726b371 | PI-W, PI-B, PI-Loss, **Div-Loss**, Uns-Act-LS | $R_7$, $R_{10}$ |
| DLT_ded6612 | **PI-W**, **PI-B**, PI-Loss, Uns-Act-LS, SD-Loss | $R_2$, $R_{10}$, $R_{14}$ |
| FCN_b170a9b | **PI-Loss**, Uns-Act-LS, Dead-ReLU, Van-Grad, Un-Fit-Batch, **SD-Loss** | $R_8$, $R_{14}$ |
| TFE_333 | **PI-W**, PI-B, PI-Loss, W-Up-Slow, SD-Loss, NR-Loss | - |
| TFE_368 | PI-W, PI-B, PI-Loss, **W-Up-Fast**, Zero-Loss | $R_{10}$ |
| TFE_742675d | Un-Sym-W, **PI-Loss(Huge Err)** | $R_8$, $R_9$, $R_{10}$ |
| TFE_bc09f95 | PI-W, PI-B, **PI-Loss(Huge Err)**, LR-Loss Un-Fit-Batch, W-Up-Slow, Van-Grad | $R_8$, $R_{10}$, $R_{14}$ |

by *TheDeepChecker*. As discussed in the paragraph 5.2.6, many training issues are correlated and induced by the same bug. In Table 5.10, based on the explanations provided by the engineers, we present the main issues reported by *TheDeepChecker* that lead them to identify and localize the root cause of the faults, whether it being caused by a coding bug or a misconfiguration.
As can be seen, most of the found faults are common (almost 96.5% of cases), which reinforces the argument that our verification routines are quite precise, regarding the problematic component and its occurring symptoms. However, given the recommended fixes from SO post's answers (see Table 5.10), we can see that the majority of fixes provided by the engineers are different. In the following, we discuss these differences in detail.

First, we observe that there are emergent fixing patterns followed by the community, which are not always efficient. Indeed, we can consider them as technical debts because they enable the convergence of training and fitting the DNN, but the main root cause of the issue is not solved, which provokes the same issue following any further changes on the DNN program or the inputs data. For instance, the buggy TF programs, *IPS-5*, *IPS-14*, *IPS-15* and *IPS-17* share the main issue of diverging loss problem, which turns its value to NaN and obstructs the training process. The initial fixes recommended by the community consists in improving the optimization routines, including the decrease of learning rate or the substitution of regular gradient-descent by advanced variants with internal adaptive learning rate like Momentum or Adam. When using *TheDeepChecker*, our two engineers were able to find the main root cause of diverging loss in buggy TF programs, *IPS-4*, *IPS-15* and *IPS-17*, which is the unnormalized inputs. In the case of *IPS-14* the problem was both the inefficient initial random weights and the poorly designed loss (i.e., using sum over the instances' errors instead of average). Without a fine-grained analysis tool like *TheDeepChecker* it was difficult for Stack Overflow users (who suggested solutions) to uncover this. The main lesson that can be derived from this example is that multiple poor design choices and coding mistakes can induce well-conditioning to the loss minimization problem, and as result, may be the origin of its divergence. Therefore, tuning the learning rate blindly will only make the training program run at its minimum capacity, and hence, the real bugs will remain hidden. By decreasing the learning rate in *IPS-15* TF program from 0.5 to 0.0005 to enable learning under the condition of unnormalized inputs, SO users only introduced a technical debt in their program. However, *TheDeepChecker* steers the engineers towards fixing permanently the root cause problem, which is the inappropriate scale of features' values. Concerning the same bug of unnormalized inputs in the TF program *IPS-17*, *TheDeepChecker* reports in the second debugging session following the normalization of the inputs that the weights is slowly updating, which lead both our two engineers to fix it by increasing further the learning rate

(i.e., both engineers ended up with taking actions that are totally the inverse of the initial recommended fix).

Second, we found that *TheDeepChecker* spots the major bug preventing the training program from fitting the model in regard to the buggy programs *IPS-4*, *IPS-11*, *IPS-12*, and *IPS-13*. Indeed, our engineers, $E_1$ and $E_2$, confirmed that the poor initial loss check alerted them to the fact that the loss is not a scalar, which led them to add the average as loss reduction strategy in *IPS-4* program. In *IPS-11*, they mentioned that the vanishing gradient problem starting from the first training iterations at the last dense layer, guides them to inspect the last layer (logits). They found that the program uses the same implemented function *fc_layer* that performs ReLU as non-linear activation for all fully connected layers. However, this useless non-linear activation erases relevant learned information and obstructs the training, because the nullified negative values make all their corresponding labels share the same probability after applying the softmax. In the case of *IPS-12*, the non-representative loss check that remains active over the iterations displaying increasingly smaller correlation, persuaded both DL engineers that either the accuracy or the loss function is mistaken, so they check them out carefully and found a typo in the accuracy function (i.e., passing logits instead of probabilities as predictions). Regarding the *IPS-13* program, the engineers adjust the learning rate and the norm penalty values to make the program satisfy the verification routines related to standard regularization risks and unstable learning of parameters that triggered, respectively, fast updated weights and overwhelming regularization loss verification routines. However, the two DL engineers failed to correctly fix the major issues contained in *IPS-1* and *IPS-7*. They proposed to change the sum reduction strategy by the average and to pass the probabilities instead of logits to the cross-entropy loss, to correct, respectively, the poor initial loss in *IPS-1* and diverging loss in *IPS-7*. However, the real bug reported by the user was the loss turning into NaN values. Because this exception is raised infrequently, the problem cannot be always detected easily. Also, *TheDeepChecker* considers the NaN loss as diverging loss and cannot provide further indications about any potential root cause. As a result of this limitation of *TheDeepChecker*, both engineers could not identify the root cause of the issue by relying on the message generated by *TheDeepChecker*, which claims that the cross-entropy loss function contains the expression $t \times \log(y)$, which renders NaN $(0 \times \log(0))$ when $t = 0$ and $y$ approaches to 0. In fact, the recommended fix was adding an epsilon ($\epsilon > 0$) to avoid the undefined expression, but a more appropriate repair for the loss numerical instability is to use, instead of hand-crafted loss, the recent TF built-in logit-based loss function, including both softmax and cross-entropy, which is numerically stable. By definition, our property-based debugging process relies on the available data and tries to catch properties' violations through watching the execution of the training program. Therefore, it

cannot detect numerical instabilities that occur in particular ranges of values. Odena and Goodfellow [119] proposed a coverage-guided fuzzing testing tool that is able to find mutated inputs triggering erroneous TF program's behaviors including NaNs raised by unstable math computation. Their evaluation shows that original and even randomly synthetic data have low chances to trigger such corner-case behaviors and expose these numerically unstable math functions. Therefore, to the best of our knowledge, fuzzy testing approaches are more suitable for detecting numerical instabilities in DNN training programs.

Besides, the results show that *TheDeepChecker* guided the DL engineers towards detecting other issues that do not prevent the program from training, but which should be addressed to improve performance and avoid all the non-optimal local minima in the loss curve. As can be seen in Table 5.10, most of these additional detected issues are related to poor initial parameters, lack of regularization, and unstable learning velocity between the layers. Nevertheless, we observed that the DL engineers proposed some different repairs to fix the same issue identified through the debugging sessions using *TheDeepChecker*. This means that there are multiple possible fixes for the same issue identified by *TheDeepChecker* and that the choice of a specific fix depends on the knowledge and experience of the engineer. Indeed, we found that for some issues, our engineers, $E_1$ and $E_2$ were able to turn off the alert and improve the performance of the training using different techniques. For instance, *TheDeepChecker* spots unstable activations with low variance regarding the first convolutional layer in the trained DNN of *IPS-1* program. Engineer $E_1$ understood that the first convolutional layer was not optimally learning the features, which led him to carefully increase the learning rate and solve the problem. In another example, Engineer $E_2$ understood that the difference in magnitude of updates between intermediate layers causes a problem of internal covariate shift, which can be solved by adding batch normalization following each intermediate layer. He went on and implemented this fix. In the future, we will examine further the fixes proposed by DL practitioners to overcome the studied training issues and analyze their impact on the quality of the code and the performance of the DNN training program.

## 5.4 Discussion

**DL knowledge remains crucial in the debugging of DNN training programs.** Many of our verification routines are implemented using statistical metrics and heuristics that are very related to inefficient training traits, so they are often connected to multiple possible root causes. In the inverse direction, most of the training pitfalls would trigger multiple fired checks because a faulty component often violates its related properties, and consequently, leads to other training properties' violations. For example, a bad initialization of weights vi-

Table 5.10 The repairs suggested by DL Engineers ($E_1$ and $E_2$) for real-world buggy TF programs

| **Program** | **Fired Checks** | **Suggested Fixes** |
|---|---|---|
| IPS-1 | PI-W, PI-b | change $W$ and $b$ initializers |
| | PI-Loss | set average instead of sum for loss reduction |
| | Uns-Act-LS | $E_1$: increase $\eta$ \| $E_2$: add batch-norms |
| IPS-4 | PI-W, PI-b | change $W$ and $b$ initializers |
| | PI-Loss | set average as loss reduction strategy |
| | Zero-Loss | add dropout layers |
| | Redundant-Layers | remove a dense layer |
| IPS-5 | Uns-Inps, Div-Loss | normalize the data |
| IPS-7 | PI-W, Miss-b | change $W$ initializers and add null $b$ |
| | Inv-Outs | add output activation layer |
| | Div-Loss | passing the probas instead of logits to the loss |
| | Sat-Sigmoid | change hidden activations (Sigmoid to ReLU) |
| IPS-11 | PI-W, PI-b | change $W$ and $b$ initializers |
| | Van-Grad (last layer) | remove ReLU on the logits |
| | Zero-Loss | add dropout for the dense layer |
| IPS-12 | Uns-Inps | normalize the data |
| | PI-W | change $W$ initializers |
| | NR-Loss | fix typo in the accuracy |
| | HF-Loss, Uns-Inference | increase the *keep_p* for dropout layers |
| IPS-13 | Uns-Inps | normalize the data |
| | PI-W | change $W$ initializers |
| | W-Up-Fast | decrease the learning rate $\eta$ |
| | Over-Reg-Loss | decrease the norm penalty $\lambda$ |
| IPS-14 | PI-W | change $W$ initializers |
| | PI-Loss | set average instead of sum for loss reduction |
| IPS-15 | Uns-Inps | normalize the data |
| | Miss-B | add null $b$ |
| IPS-17 | Uns-Inps | normalize the data |
| | W-Up-Slow | increase the learning rate $\eta$ |

olates the required asymmetry between neurons, however, the resulting unbreaking symmetry would lead to other issues, like over-negative weights, Dead ReLUs, and vanishing gradients. Indeed, all the neurons will receive identical gradients and evolve throughout training, effectively preventing different neurons from learning different things. Thus, it is likely that a non-optimal gradient update, from the starting iterations, would be applied symmetrically to all the neurons, and consequently, would cause the stagnation of the neural network.
*TheDeepChecker* incorporates dynamic verifications with periodic inspection reports, narrowing down the space of suspicious states, which help the user recognize fault patterns and identify the root cause by analyzing the timeline of fired checks (i.e. their chronological order) and the reported information (i.e., positions, metrics, thresholds, etc.). Nevertheless, these mitigation strategies require sufficient DL knowledge and skills. Developers need to be sensitive to such details in order to be able to efficiently interpret the debugging reports.

**Scoping on Feedforward Neural Network Architecture.** Many of the targeted training pitfalls and the proposed properties are generalizable to other model architectures [220], but in this paper, we focus on their application for the feedforward architecture, which is, first, widely used in several regression and classification problems, as well as, reinforcement learning tasks. Second, it is the basic neural network model that influences novel architectures, and even represents one of their building blocks. Nonetheless, recurrent neural networks (RNNs) have faced more severe gradient problems [221], including vanishing and exploding phenomena. Generative adversarial networks (GANs), which particularly leverage two on-training models in a min-max game, raise novel training issues in relation to the learning stability and convergence, in addition to other GAN-specific problems [222–224] such as mode collapse, where the generator outperforms quickly the discriminator, without fitting the data distribution, but through simply rotating over few data types.

**Usability Study Limited on the Mappings from Checks to Fixes.** Given the real-world DL programs published by SO users, we recruit two DL engineers from two different teams and having different backgrounds for a usability evaluation, focusing on the mappings from the fired checks to the DL program fixes. Nonetheless, other important dimensions such as the time spent on debugging and the relevance of the proposed fixes, could be assessed in comparison with the trial-and-error debugging process without *TheDeepChecker*.

### 5.5 Threats to Validity

**Selection Bias.** The selection of the subject DL training programs could be an internal threat to validity. In this paper, we try to counter this issue by using two complementary empirical evaluations on synthetic buggy programs and real buggy programs. We used diverse

base DL training programs that solve regression and classification problems. They encode DL models with different architectures, complexities, and techniques. Moreover, the base synthetic DL programs have official implementation references and run on widely-studied datasets (i.e., Auto-MPG, MNIST, CIFAR10). Then, the buggy versions of these synthetic DL training programs were created to mimic the DL faults reported and studied in empirical studies on DL programs' defects. The discussion on the results of the evaluation on the synthetic buggy programs provides insights on how the properties and heuristics deployed in the verification routines were able to detect the behavioral training issues caused by the injected bug. Indeed, we leveraged totally-disconnected workflows for the construction of the synthetic buggy programs and the verification routines. Figure shows how the abstraction of DL faults to synthetize buggy subjects was done essentially on former empirical studies' datasets and our manual inspection of TF-related SO posts. On the other hand, Figure shows how the design of verification routines was guided by applied DL research works and technical DL expert reports about troubleshooting to codify fundamental properties of DL programs and practical heuristics on proper DL training dynamics. It was crucial to keep synthetic bugs and verification routines separate in order to avoid hard-coded verifications that target the mainstream DL faults identified by the community. Although the injection of DL faults was done on reference DL models with minimal code changes, it is still a human-crafted process that may contain imperfections. Furthermore, each synthetic buggy program represents a well-designed base program with only a single bug injected. Thus, we complement the evaluation using 20 real-world buggy programs from the dataset provided by Zhang et al., which are related to the scope of our targeted DL bugs, and represent more realistic conditions of debugging since they may contain different issues simultaneously, and even issues that are not uncovered by maintainers yet.

**Settings' Generalizability and Transferability.** The setup of heuristic-based thresholds could be an external threat to validity. In our design and implementation of property-based verification routines, we relied on the original documentation sources [24, 177, 179, 202] that described the issues, to set up the thresholds, when they are indicated. Nonetheless, some properties were always presented and studied through visual plots comprehensible by humans. This makes the design of metrics and their thresholds challenging, but we focus on the abstract violation traits of the properties rather than the concrete studied instances, in order to be able to construct a verification routine based on the foreknown training misbehaviors, as discussed in the implementation strategies (section 5.2.2). Therefore, given the dynamic and continuous aspects of *TheDeepChecker*, it was possible to set up intuitively pessimistic thresholds, to be our default configuration, in order to ensure a high coverage with less false alarms when enough monitored training iterations were executed. Indeed, we

avoided the empirical tuning of these thresholds because we did not have access to a large benchmark of reproducible buggy programs. Besides, the shrinking of suspicious program state includes the computed metrics and thresholds that were behind the fired checks, which is a mitigation strategy to help the user ignore false positives. Given the variables' instatiations in the violated rules, users can also assess the sensitivity of the thresholds on their DL application domain, which allows them to set up a more precise custom configuration for *TheDeepChecker*.

## 5.6 Chapter Summary

This chapter reports about the design and implementation of *TheDeepChecker*, an end-to-end automated debugging approach for DNN training programs. To develop *TheDeepChecker*, we systematically gather a catalog of pitfalls commonly occurring in the development of DNN training programs. Then, we explore various resources on applied DL researches and technical reports with aim of distilling fundamental properties and practical heuristics that can be codified into verification routines to detect the DL pitfalls' resulting faults and training issues. Next, we develop a property-based debugging approach, named *TheDeepChecker*, that orchestrates the different properties' verification over multiple phases. On the one hand, we evaluate *TheDeepChecker* on synthetic buggy programs that contain each an injected DL fault. The results show its effectiveness at detecting DL coding bugs and misconfigurations with (precision, recall), respectively, equal to (90%, 96.4%) and (77%, 83.3%). Moreover, we compare *TheDeepChecker* with Amazaon Sagemaker Rule-based Debugger(*SMD*) on real-world buggy programs extracted from SO and GH. The results show that *TheDeepChecker* outperforms *SMD* by detecting 75% rather than 60% of the total of reported bugs in the SO post accepted answer or the bug-fixing commit message. Indeed, *TheDeepChecker* effectively captured the slightest violation of all mandatory training assumptions, even those having only a minor negative effects on the training process, providing sufficient feedback on any problematic issue in DNN program. Using *TheDeepChecker*, two DL engineers were able to successfully locate and fix 93.33% of bugs contained in 10 buggy TF programs.

# CHAPTER 6 DEEPEVOLUTION: SEARCH-BASED DL TESTING

Although DL software has proven useful and effective in many fields, their growing adoption in massive and critical systems accentuates the urgent need for advanced DL testing methods to address the blindspots of iid performance evaluations, and hence, mitigate the risks of DNN behavioral shifts in production. Nowadays, developing these methods has become a focal point in both academia and industry. Coverage-guided fuzzing [119] expands the iid evaluations by injecting arbitrary noises in a corpus of test data, triggering uncovered state of activations, in order to reveal the DNN weaknesses. However, modern DNNs are sufficiently smooth and regularized to make correct predictions despite these random noisy inputs. Adversarial attacks [13] can alter maliciously the original inputs, leading to unavoidable mispredictions, however, they are designed to test the security vulnerabilities of the DNNs with intentionally crafted adversarial examples (AX) that are often weakly-correlated with real-world corner case scenarios. Contrarily, the challenge of DL testing is to verify whether a DNN can be trusted in practice, i.e., under normal and extreme conditions required by an application. In that regard, metamorphic testing (MT) probes a trained DNN by observing its outputs on specifically designed inputs that express the desired requirements, as explicit statements of expected DNN behaviors under certain input transformations. In computer-vision field, several MT-based approaches [16] have been proposed to systematically generate synthetic inputs that test the robustness of a convolutional neural network(CNN) against derived corner cases. However, their inherent data generators are already limited, either in terms of the variability of the input mutations or the efficiency of the optimization process. In fact, the first-order gradient methods [17, 20] efficiently carry out pixel transformations by varying their values based on prominent directions of the gradient, but cannot handle complex image transformations such as rotation or random erasing. Even though greedy search-based methods [18, 19] can handle diverse input transformations, they use simple strategies to drive input generation, such as mutating previously discovered high-potential inputs rather than sampling new ones, which cannot guarantee optimal or near-optimal results.

In this vein, we propose DeepEvolution, a search-based approach for testing DL systems across many domains. DeepEvolution evaluates that a DNN is robust and would achieve stable and sustained high accuracy under different usage scenarios that are not directly guaranteed by iid evaluations. First, we develop MRs relying on domain-specific data transformations, iterative parameters tuning, as well as post-transformation validity checks. The MRs enable the specification of the desired system requirements in terms of must-be-held mappings between input distortions and their specific deviations on the outputs. DeepEvolu-

tion encodes the metamorphic transformations into a data-independent constrained space of their vectorized parameters and settings. Second, DeepEvolution relies on population-based metaheuristics algorithms and behavioral drift fitness functions to steer the search over the space of transformations towards the most fault-detecting regions relative to the test objective, as follows: (i) exposing misclassified inputs to probe for coincidental and spurious associations, and assess whether the optimized DNN encodes appropriate inductive biases that are robust to naturally-occurring noises and irrelevant property changes; (ii) detecting difference-inducing inputs to probe for quantization defects and low-precision inefficiencies, and test whether the compressed DNN preserves the learned patterns that are relevant to generalize as expected in the deployment settings.

The effectiveness of DeepEvolution was assessed using a variety of case studies, including visual, speech, and natural language text recognition DNNs applied to popular datasets including CIFAR100 [225], IMAGENET [143], KWS [226], RAVDESS [227], and IMDB [228]. Results show that DeepEvolution succeeds in revealing DNN erroneous behaviors. Precisely, it achieves, averagely, 41%, 24.5%, and 5% of misclassification detection rates for the studied DL domains, respectively, image, audio, and text. DeepEvolution was also able to expose hidden quantization inefficiencies. Specifically, it reaches, on average, 21.5%, 24%, and 2% of divergence exposure rates when comparing genuine and quantized DNNs designed for visual, speech, and natural language text recognition. Throughout all the studied cases, DeepEvolution outperformed TensorFuzz which is Google Brain's coverage-guided fuzzing framework specialized for DNNs.

**Chapter Overview.** Section 6.1 describes DeepEvolution, its design process, and its main components. Section 6.2 reports the study cases and their associated evaluation results, while Section 6.3 discusses the threats to validity of our experiments. Finally, Section 6.4 summarizes the chapter.

## 6.1 Approach

### 6.1.1 Problem Formulation and Proposed Solution

Assuming we have an original test dataset $D_{test} = (x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$ and $f$ is the classification neural network under assessment, we design a space of semantic-preserving metamorphic transformations, $\mathcal{T}$, where each included metamorphic transformation, $t_\theta \in \mathcal{T}$, represents a composite input transformation containing multiple parametric distortions and their parameters $\theta$ are within the tuned valid boundary space, $\theta \in \Theta$, which ensures the preservation of the semantic input identity after applying the distortions. A population-

based metaheuristic algorithm iteratively explores the space of transformations, returning a set of high-potential valid candidates that are likely to transform the original test data into new synthetic inputs exhibiting erroneous DNN behaviors. Indeed, the metaheuristic searching algorithm maximizes the scores of the generated synthetic test inputs, $\hat{x}_i = t_\theta(x_i)$, w.r.t the selected test adequacy criterion, $C$, while penalizing those that are close-to or violate the validity constraint, $\Phi_T(\hat{X}) > thresh$, which ensures that distorted input retains the semantic identity of its source entry. The formulation of the maximization problem can be as follows:

$$\underset{\theta}{\text{maximize}}\ t_\theta(X) \text{ subject to } \Phi_T(\hat{X}) > thresh,\ \hat{X} = t_\theta(X) \tag{6.1}$$

As the search iterations progressed, the follow-up test asserts the predictions to detect the target failures. For robustness assessment, the assertion consists of verifying the equation: $f(t_\theta(x_i)) = f(x_i)$. For quantization evaluation, the assertion requires verifying the equation $f(t_\theta(x_i)) = f_{original}(x_i)$, where $f$ is the quantized neural network under test and $f_{original}$ is its full-precision counterpart. All the distorted inputs $\hat{x}_i$, leading to failed assertion, are tracked and stored to compute overall assessment metrics and provide further insights.

### 6.1.2 Semantically-preserving Metamorphic Transformations

To define the semantically-preserving metamorphic transformations for different data types, we require two acceptance criteria for each input transformation: (i) it should be parametric to enhance the variability of the produced synthetic inputs and the level of induced distortions can be controlled through pre-fixing the range of its parameters; (ii) it should be deterministic, i.e., returns the same synthetic input $\hat{x}$ given the same parameters $\theta$. Due to this determinism, the metaheuristic algorithm would get stable fitness values when evaluating candidates from the transformation search space, which is crucial for the success of the searching process. In the following, we list the used metamorphic transformations for each supported data type, and we describe the process of tuning their valid range of parameters and their associated validity checks.

#### Image Transformation

The image transformations can be organized in the following two groups:

**Pixel-value Transformations:** Firstly, we can adjust the contrast, brightness, blur, sharpness, and color intensity of the image. Those image mutations are applied to all pixels of an image, so their common parameter is a floating-point factor that controls the

strength of the induced effect, for example, a lower value means less brightness, contrast, etc., and a higher value means more. Secondly, we have included random erasing, which can be viewed as dropping out contiguous areas of a given image in order to simulate arbitrary occlusion, i.e., when parts of an object are unclear. The location of the masked area and its associated pixel value are set up as parameters. Last, we model two types of random pixel perturbations as follows: (i) constrained additive noise: the parameters are the pixel locations to alter and their associated additive noises; (ii) salt and pepper noise: the parameters are the location of salt pixels, i.e., turns into white ones, and the location of pepper pixels, i.e., turns into black ones.

**Affine Transformations:** We implement image translation, scaling, and shearing that accepts two parameters corresponding to the geometric perturbation level with respect to the spatial coordinates ($x$ and $y$). In addition, we also include the image rotation that takes an angle $\theta$ as a parameter.

### Audio Transformation

There are also two main categories of audio transformations, as follows:

**Full-Signal Property Alterations:** We include time stretching the signal up or down without altering the pitch, pitch shifting the sound up or down without altering the tempo, and changing the loudness of the sound. Each of these sound mutations operates on the entire signal and takes a floating-point factor to determine the direction and the level of the mutation.

**Window-based Signal Perturbations:** First, we introduce several injectors of colored noises (white/pink/brown) that have different properties. A colored noise signal is added based on three parameters: its length, its starting timestep, and a floating-point factor that controls its amplitude relative to its original value. To simulate indoor and outdoor background noises, we collected 11 samples of environmental sounds including footsteps, mouse clicks, and rain, etc. Injecting environmental noise is done with the same logic and parameters as injecting colored noise with the addition of the selected noise index. Finally, we define a constrained signal perturbation, which can be configured with the indices of timesteps to alter, as well as, the additive noise value for each altered timestep.

### Text Transformation

We separate the parametric text perturbations into the following two types:

**Character-level Perturbations:** We use character insertion, swapping, and deletion, which modify one or more words by deliberately creating misspelled variants of them. Fine-grained text transformations are analogous to imperceptible noise in signal data because the original sentence and its transformed counterpart appear visually or morphologically similar to humans. The parameters they accept are the selected word positions as well as the indices of the involved characters.

**Word-level Transformations:** We first consider word deletion and word swap, which require, respectively, the index of the word to be eliminated and the indices of the two words to be swapped. Secondly, we incorporate word insertion, synonym replacement, and embedding replacement, all of which rely on a word dictionary and a distance metric to find a synonym or a word that has a close embedding to a given word in the sentence. An additional specific word index is required to select the replacement word to choose from the returned list of nearest neighbors, based on embeddings or natural language dictionaries.

### Bounding Ranges of Transformations' Parameters

For each transformation's parameters, the range of possible values determines the degree of distortion induced in the input. Indeed, a wide range may include extreme values that would result in high loss of information, leading to meaningless inputs. Inversely, a narrow range would be too conservative, leading to only the meaningful inputs that are very close to their source data. Hence, we tune these parameter ranges beforehand to guarantee a good balance between enhancing the diversity and preserving the semantic identity of the transformed inputs. To do so, we use appropriate validity scores to determine how much information is lost after the distortion, as described in the next Section 6.1.2. Thereby, we first start with the full range of possible values, then narrow down the range if the validation scores of the resulting synthetic inputs are statistically low and re-iterate again; otherwise, we stop. For each candidate range, we apply the transformation with sampled parameters on arbitrary original instances to infer the valid ratio of the resulting transformed versions.

### Post-transformation Validity Check

In spite of tuning the ranges of each distortion's parameters, there is no guarantee that the synthetic inputs would preserve the semantic identity of their genuine parents, especially when multiple distortions are applied as a composite one. In fact, it is probable that their application at once to the input could lead to meaningless inputs, even if each distortion is

separately verified to be semantic preserving under the condition of keeping the parameters within the tuned set of valid ranges. Thus, we set up a post-distortion validity test where we use a data type-dependent score and its associated threshold to discard meaningless inputs. In the following, we detail the implemented scores, representing appropriate similarity measures that determine if the transformed input is still semantically equivalent to the original parent after applying all the active distortions.

**Image validation:** We compute the Structural Similarity Index (SSIM) [229] that returns the similarity between the two images ranging from 0 to 1.0 based on the visual impact of three characteristics: luminance, contrast, and structure. In order to ensure that the transformed images remain semantically equivalent to their original sources, we discard the transformed images for which the SSIM values of their comparison with their sources fall below 0.8.

**Audio validation:** We calculate the signal-to-noise ratio (SNR), which is a measure of the strength of the desired signal relative to background noise (undesired signal). The ratio is expressed as a single numeric value in decibels (dB). An SNR over 0 dB indicates that the signal level is greater than the noise level. Then, the higher the ratio, the better the signal quality. In our distorted audio verification, we adopt the threshold of 20 dB as suggested in [230, 231], in order to systematically separate the valid synthetic sounds (SNR$\geqslant$20) and invalid ones (SNR$<$20).

**Text validation:** We keep track of all the edits applied to the original text over the composite distortion. Then, we define the maximum percentage of allowed edits to the sentence to be 25%, as suggested in Alzantot et al. [232]. Thus, all the generated transformations that lead to a higher percentage of edits, will be considered as invalid, and vice versa.

### 6.1.3 Search-based Test Generation Approach

#### Search Space: Transformation Vector Encoding

Due to the diversity and complexity of our codified input transformations, the space of distorted neighbors of an original datapoint is challenging to define, contrary to vanilla constrained input perturbations that would set up the search space for an input $x$ to be $[x - \delta, x + \delta]$, where $\delta$ is the maximum allowed value deviation. Hence, we chose to search for transformations instead of data instances. To that end, we encode the parameters of each single data transformation into a sub-vector. Then, the vector associated with the composite metamorphic transformation is the result of the concatenation of all the sub-vectors

obtained from the transformations included, along with binary variables to activate or deactivate each one of them. Based on the tuned range of parameters for all the included input distortions, we can create high and low boundary vectors, bounding the search space of composite metamorphic transformations in a way that increases the chances of finding valid candidates. Due to the high dimensionality of the input space, the transition from input space to lower-dimensioned transformation space simplifies the search problem by narrowing the search space as well.

**Behavioral Drift Fitness Function**

Considering the composite transformation space, $T$, to be searched, the optimization needs a fitness/objective function, $f_T$, that compare and contrast the fault-revealing capabilities of transformations, $t \in T$, if applied to the original input, x. Nevertheless, our main goal is to reveal the maximum of diverse failed test inputs; as a result, we design a fitness function, $f_X$, that measures how much behavioral deviations the resulting transformed input, $\hat{x}$, causes in the DNN under test. Thus, the connection between both fitness functions, $f_X$ and $f_T$, would simply be $f_X(\hat{x}) = f_T(t)$, where $\hat{x} = t(x)$. Furthermore, we are interested in classification neural networks with different architctures that have been applied in computer vision, speech recognition, and natural language processing. Given a $c$-classification problem, a neural network should define a last dense layer with neurons equal to the labels count, called logits, $\boldsymbol{l} = (l_1, ..., l_c) \in \mathbb{R}^c$. Then, an adjacent softmax activation layer normalizes the logit scores into a probability distribution, $\boldsymbol{s} = (s_1, ..., s_c) \in \mathbb{R}^c$, whose component is the probability of a class label membership and is computed as follows, $s_i = \sigma(l_i) = \frac{e^{l_i}}{\sum_{j=1}^{c} e^{l_j}} \quad for \quad i = 1, ..., c.$ Hence, the predicted label for the input is the one with the highest probability, which can be formulated as follows:$\hat{y} = argmax(s)$.

While adversarial attacks minimze the correct label's score or maximize another target's score, we seek to capture all the fine-grained divergences in the softmax score distributions to uncover erroneous behaviors, while enhancing diversity and avoiding overuse of the discovered areas of high potential. To that end, we find two common divergence measures to compare different probability distributions, Kullback–Leibler divergence ($KLD$) [233] and Jensen–Shannon divergence ($JSD$) [234], and we opt for Jensen–Shannon divergence, denoted $J$ in the defintion of our fitness functions. Let $Q$ and $R$, be two probability distributions defined on the same space $\chi$, where ($|\chi| = c$), we can compute $J$ as follows:

$$J(Q||R) = \frac{1}{2}(D(Q||M) + D(R||M)) \tag{6.2}$$

where $D(Q||R) = \sum_{i=1}^{c} Q(i) \ln(\frac{Q(i)}{R(i)})$ and $M = \frac{1}{2}(Q + R)$. The main reason we chose $JSD$

over $KLD$ is that it is symmetric, $J(Q||R) = J(R||Q)$, and bounded, $0 \leq J(Q||R) \leq 1$.

**Robustness Assessment.** The objective is to estimate the divergence between the two class membership probability distributions obtained by the DNN under test for the original input, $x$, and its transformed descendent, $\hat{x}$. Moreover, maximizing this divergence would continually widen the discrepancy between the model responses for the two inputs in all directions, which may lead to a mismatch in their assigned labels (i.e., the most probable one given the probabilities of class memberships). Considering $s(x)$ to be the softmax activation layer for a given input $x$ and $S_{valid}$ is the normalized validation score of $\hat{x}$ w.r.t $x$ within $[0, 1]$, our behavioral drift fitness function for a synthetically-produced input can be formulated as follows:

$$f_{\mathcal{D}}(\hat{x}) = S_{valid} \times J(\boldsymbol{s}(\hat{x}), \boldsymbol{s}(x)) \tag{6.3}$$

**Quantization Assessment.** The goal is to determine the difference between the two class membership probability distributions obtained by the original and quantized DNNs. Therefore, maximization of this divergence measurement would increase the likelihood of disagreement between the two models' responses, until eventually reaching a mismatch between their two predicted labels, i.e., their individual most probable labels for the same input. Let $m_o$ and $m_q$ be the original model and its quantized version, $S_{valid}$ is the normalized validation score of $\hat{x}$ w.r.t $x$ within $[0, 1]$, as well as $\boldsymbol{s}_o$ and $\boldsymbol{s}_q$ be their respective softmax activation functions, our behavioral drift fitness function for a synthetically-produced input can be formulated as follows:

$$f_D(\hat{x}) = S_{valid} \times J(\boldsymbol{s}_o(\hat{x}), \boldsymbol{s}_q(\hat{x})) \tag{6.4}$$

In our approach, we support two modes. The first is the single instance mode where only one original input $x^i$ is fixed in the loop, so that all the generated transformations $t^j \in T$ would be evaluated based on the fitness $f_T(t^j) = f_X(\hat{x}^{i,j})$, where $\hat{x}^{i,j} = t^j(x^i)$. The second is the batch mode that fixes a subset of $B$ original inputs, $X_b$, at once, so that any data transformation $t^j \in T$ would be evaluated based on the average fitness $\bar{f}_T(t^j) = \frac{\sum_{i=0}^{B} f_X(\hat{x}^{i,j})}{B}$, where $\hat{x}^{i,j} = t^j(x^i), \quad \forall i \in [0, B]$. As explained above, we are exploring the search space for better coverage of potential failures and avoiding mode collapse, where the input generation exploits one weakness of the model to reveal slightly-different failures based on the same real data. For the sake of simplicity, we have formulated the above-mentioned fitness functions in the single instance mode.

**Metaheuristic searching algorithms**

DeepEvolution is a search-based DL software testing approach that leverages metaheuristics optimization techniques to produce test inputs with high fault-revealing ability. Specifically, we use nature-inspired population-based metaheuristics that feature complex routines and intrinsic non-determinism, making them suitable for identifying vulnerable regions in the large, multi-dimensional input space of the DL models. Several researchers have employed them in crafting adversarial examples under black-box settings and have shown their effectiveness in the areas of our interest, including computer-vision [49–52], natural language processing [53–55], and speech assistance DNNs [56]. Concretely, we implement 1 evolution-based algorithm and 8 swarm-based algorithms. In these implemented metaheuristics, all the individuals of a generation are assessed before their updates are inferred. As a result, they are effective when it comes to DL software testing because DNNs are designed to predict for a batch of inputs simultaneously; so, all the required softmax probabilities are requested at once, the fitness values are computed, and the updates are determined. In line with the No Free Lunch Theorem (NFL) [235], we investigate various nature-inspired, population-based metaheuristics because none of them outperforms all others for all possible classes of optimization problems. In the following, we introduce our implementation of the selected algorithms. On the first hand, genetic algorithm (GA) [58] is the most popular evolution-based method that mimics the behavior of biological evolution, including the natural selection and reproduction, i.e., the fittest survive and reproduce. To generate stronger individuals in every generation, GA performs the following operations: (i) selection: we sort the individuals w.r.t their fitness values, then, we select the top k of them to be the parents for the next generation; (ii) crossover: several binary crossovers for breeding are supported such as one-point [59], two-point [59], and uniform [59], which is our default option, and (iii) mutation: we use a random walk with a small step size to move the offspring transformations arbitrarily from their inferred positions, depending on a certain mutation probability. On the other hand, swarm-based methods imitate the dynamics of natural swarms, (i.e., group of animals such as flocks of birds or gray wolves), especially, how members of the swarm interact with one another and with their environment. The following established swarm-based algorithms are included: (1) Particle Swarm Optimization (PSO) [57], (2) Firefly Algorithm (FFA) [236], (3) Gray Wolf Optimizer (GWO) [237], (4) Moth Flame Optimizer (MFO) [238], (5) Whale Optimization Algorithm (WOA) [239], (6) Multi-Verse Optimizer (MVO) [240], (7) Salp Swarm Algorithm (SSA) [241]. These selected metaheuristic algorithms perform their distinct subtle steps, including stochastic, diversity and selection to trade-off between the intensification (exploiting the results and concentrating the search on regions near effective solutions found) and diversification (exploring non-visited regions to avoid missing interesting potential so-

lutions) [242]. Their generic-purpose design enables them to be used for a wide range of constrained optimization problems involving high-dimensional, bounded real-valued vectors without prior discretization. Therefore, all these nature-inspired metaheuristics enhance the fitness values of the evolving population of candidate transformations, over iterations, resulting in synthetic test inputs that are more likely to cause deviations from the expected behavior, and thus more likely to expose DNN's weaknesses. Along with the increased fitness, both diversification and intensification mechanisms that these metaheuristics employ for searching the fittest candidates, produce new inputs sufficiently different from the old ones to uncover regions of interest while also being similar enough to inputs that have high fitness values to uncover more failures in the previously-discovered regions. In our test generation problem, we do not seek an optimal or suboptimal solution that would represent the input transformation, leading to the test input with the highest fitness. Hence, we slightly modify the standard design of population-based metaheuristics to continuously monitor the fault-revealing transformations (i.e., yielding AXs) among evolving feasible solutions (i.e., meet all the validity constraints).

**Narrow Down the Search Space**

For all the supported data types, we include various switchable parametric data transformations to form our search space of composite distortions. This high-dimensional transformation space enhances the diversity of the synthetic test data compared to approaches that concentrate on one type of distortion and refine searches over its parameters to find the optimal settings. However, this dimensionality comes with challenges in regards to the test input generation. A huge space of composite distortions can make it difficult for the optimization to find valid combinations. The post validation is a data-dependent process, measuring the distance of the distorted data from its source, hence, in some cases built-in transformations can be aggressive, systematically leading to invalid synthetic inputs that are always rejected. In fact, we aim to design a periodic narrow-down of the search space that eliminates the least relevant transformation at every period of iterations. To infer the relevance score, we build a Bayesian probabilistic scoring method that applies the bayes rule to update, over the iterations, the likelihood of each transformation being part of a valid and 'adequate' composite transformation. First, we define a binary random variable that is true when the composite distortion yields valid and adequate test input, i.e., its validity score is higher than the predefined threshold and its fitness is higher than the median; otherwise, it is false. Then, we consider the switch variable as a binary random variable to indicate if a transformation is either active or inactive when a composite distortion is applied. Thus, the relevance score of an included transformation is the probability of having a valid adequate distortion given that

the underlying transformation is active. Using composite distortions tracked over iterations, we estimate the joint probability of a valid adequate distortion and a given transformation being active, then, we normalize it by the marginal probability of having a valid adequate distortion in general to derive the relevance score. Furthermore, we may encounter the 'Zero conditional probability Problem' if the underlying transformation was never active or no composite distortion was valid and adequate by accident. To avoid returning zero probabilities, we leverage laplacian correction to define a neutral probability in such cases with the aim of preserving the score-based ranks of the evaluated transformations. Last, we sort the transformations by their inferred relevance scores, then, we remove the one having the lowest score, i.e., it was less-frequently active in the obtained valid and adequate candidates so far.

### 6.1.4 DeepEvolution: Workflow and Algorithm

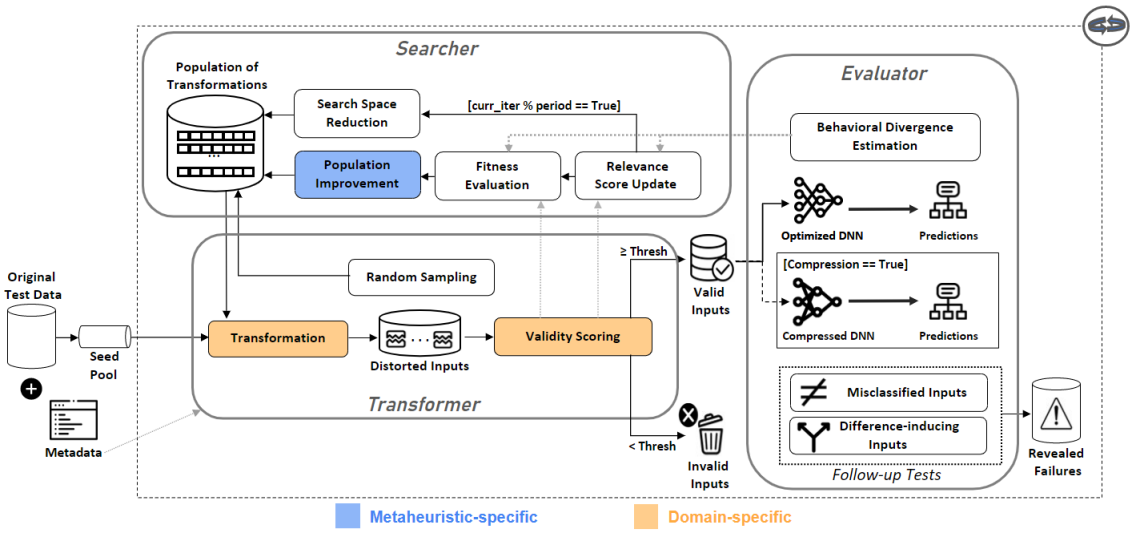



Figure 6.1 Overview design of DeepEvolution

Figure 6.1 presents an overview of the design of DeepEvolution which is composed of the following components.

**Transformer:** It contains the processing logic of the metamorphic transformations that can be configured based on the metadata, including the data shapes, the data range, and the tuned set of valid parameters. Moreover, it exposes an interface providing multiple operations to create a random metamorphic transformation, to check transformation's element bounds and clip them within the valid range, and to apply the transformation on a given input data. In addition, the transformer includes a validity score function,

depending on the data type, which estimates the degree of distortion on the input post-transformation compared to its original parent.

**Evaluator:** It is responsible for handling the one or more DNNs involved in the evaluation process. When necessary, it gets the predicted labels, probabilities, and even intermediate activations, then, it enables the assessment of different distance/divergence measures to capture how much a given synthetic input is able to yield a DNN's response distant from its source data, or to cause a divergent behavior between the original model and its quantized counterpart.

**Searcher:** It uses a population-based metaheuristic algorithm to explore the space of transformations towards prominent directions where there are subtle and interesting transformations that have high chances to expose erroneous behaviors. Below, we introduce the main four sub-components that make up our searching method:

*1. Population Initializer:* It initializes randomly a set of valid candidate transformations.

*2. Fitness Evaluator:* It computes the fitness of a given candidate transformation with respect to the established function according to the assessed quality attribute.

*3. Population Updater:* It encapsulates the selected metaheuristic strategy to infer the next population in a way that all the individuals fall within the tuned ranges and they are likely to be stronger and more fit than their predecessors.

*4. Space Reducer:* It periodically removes the least relevant transformations from the search space in order to keep the searcher focused on high potential areas.

---

```
import Searcher, Evaluator, Transformer

# D: correctly classified input seed,
# m_o: original model and m_q: quantized model
D, m_o, m_q = load_ingredients(model_config)
N_seeds, seed_size = load_params(data_config)
narrow_down_period, max_iter, pop_size = load_params(search_config)
# if m_q is None, check for Robustness; else check for Divergence
Evaluator.set_up_models(m_o, m_q):
# initialize the set of failed transformation vectors
T_fails = set()
for X, y in data.loader(D).sample_seeds(N_seeds, seed_size):
    # initialize random population T of transformation vectors
```

```
T = Searcher.init_population(pop_size)
iteration = 0
while iteration < maxiter:
    # apply every t in T on X
    X_trf = Transformer.transform(T, X)
    # compute the validity scores S for all the transformed inputs
    # and return the set of valid inputs X_valid
    X_valid, S = Transformer.validity_check(X_trf, X)
    for x_d in X_valid:
        # check if x_d breaks the MR
        if Evaluator.is_failed(x_d, y, m_o, m_q):
            # t_d is the transformation vector leading to x_d
            T_fails.add(t_d)
    # compute the probability divergences induced by each x_trf
    J = Evaluator.compute_proba_divergences(X_trf)
    # update the relevance score for each transformation
    Searcher.update_relevance_score(T, J, S)
    # reduce the search space in each narrow_down_period
    if iteration % narrow_down_period == 0:
        Searcher.narrow_down_search_space(T, J, S)
    # update the population of transformations T
    T = Searcher.improve(T, J, S)
    iteration += 1
```

Pseudo-Code 6.1 DeepEvolution Test Driver Algorithm

The proposed DL software testing algorithm is presented in the pseudo code 7.1. First, the original test data is loaded and filtered to keep only the test inputs that are correctly classified by the DNN. Second, the searcher initializes an initial random set of valid metamorphic transformations (the size of the population is the first common parameter). Taking a random seed of filtered inputs, the transformer applies the sampled transformations on the inputs to produce the synthetic test inputs, then, the latter passes the validation test to eliminate the meaningless inputs caused by an accidental high loss of information. Next, the evaluator runs the one or more DNNs under test to classify the valid synthetic inputs, and to compute the induced distribution divergences based on their respective label probabilities. Then, it runs the follow-up test assertions on the obtained synthetic test inputs' predictions in order to detect potential erroneous behaviors, including DNN weaknesses by checking if the predictions differ from the original actual labels, or quantization inefficiencies by comparing

the predictions of the original DNN and its quantized counterpart. All the fault-revealing transformations that were capable of producing failed test inputs, are stored for global insights and further analyses. Afterwards, the searcher utilizes the validity scores and the distribution divergences to compute the fitness values of each individual transformation and update their relevance scores. Periodically, the search space is reduced by eliminating the least relevant transformation in the remaining optimization iterations for the current seed of inputs. Based on their obtained fitness values, the population of transformations is updated following its inherent metaheuristic strategy, while keeping them within the valid boundaries. Finally, the transformer applies the newly-generated transformations on the seed of inputs, then, the process repeats the same following steps to produce the next generations, until reaching the fixed maximum number of iterations (the second common parameter). Therefore, the next seed of original test inputs is sampled, the search process restarts from the beginning with a full transformation space and keeps tracking all the failed tests.

## 6.2 Evaluation

### 6.2.1 Experimental Setup

In this section, we detail the different elements of our experimental setup.

#### Datasets and Pretrained DNNs

**Image.** CIFAR-100 [225] is a labeled dataset of 60000 colored images of 32×32 pixels, which can be grouped into 100 classes. We use the EfficientNet-B0, which contains a simple EfficientNet architecture with 9 convolutional layers. EfficientNet is a new class of CNN's built by Google. These CNNs provide an effective compound scaling method to achieve maximum accuracy gains with reduced number of parameters as compared to the other state-of-the-art CNNs like ResNet. Imagenet [143] is the most popular dataset of object recognition, containing 1 million images of 224x224 pixels, which can be classified into 1000 labels. We use Vision Transformer (ViT) which is a transformer encoder-decoder model (BERT-like) that transforms the images into a sequence of fixed-size patches (resolution 16x16), which are linearly embedded as absolute position embeddings. As a result, the pre-trained transformer can then be used to extract the useful features needed to train an accurate classification network downstream.

**Audio.** KWS Speech Commands [226] is a popular dataset of speech recognition, assembling 65,000 clips of one-second duration. Each recorded clip corresponds to one of 30 possible keywords, where only ten of them are commands used in a robotics environment and others

represent noise words. We tested the end-to-end model proposed by Google that consists of a speech feature extraction based on MFCC and a neural network based classifier. Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [227], containing 7356 audio files. The classified emotions are neutral, calm, happy, sad, angry, fearful, surprise, and disgust. Audio data are first preprocessed to extract fixed-length input features based on MFCC [243]. Then, we use CNN proposed by depinto et al. that defines 1-D convolutional layers with Relu activations, max-pooling layers with filters of size 2, and then, the dense layer with a softmax activation.

**Text.** IMDB [228] is a well-known dataset for binary sentiment classification, containing 50k of highly-polarized movie reviews. The review sentences are first projected into a fixed 300-dimensional embedding space using GloVe [244]. Then, we evaluate two different DNNs in solving the sentiment analysis classification task, including Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). First, WordCNN defines 1-D convolutional layers with 250 filters of 3 size, max-pooling layers, and a dense layer with a softmax activation at the end. Second, BiLSTM comprises bi-directional LSTM layers with 128 and 64 computational units in both forward and backward passes, and finally a dense layer with a softmax activation.

### Quantization

As a matter of credibility and reproducibility, we opt for Tensorflow Lite [1], Google's open-source DL library to assess the state-of-practice quantization for on-device deployment. We quantize each original DNN with 32-bit arithmetic precision using the dynamic range post-training quantization technique. The latter compresses the DNNs by reducing the precision of their weights without any further retraining. In our study cases we use it to statistically transform all the weights into integer data type of 8-bits precision, then, these weights are reconverted to their original floating-point data type to compute the activations at inference. This data type conversion is done once and cached to reduce latency at inference.

### Baseline

We use TensorFuzz [119] as a competitive baseline, which is an open-source Tensorflow-based framework that implements a coverage-guided fuzzing approach specialized for DNNs. It consists of handling an input corpus that evolves through the execution of tests by applying random mutation operations on its contained data and keeping only interesting instances that allow triggering new activation traces. The novelty of activation traces is identified based

[1] `https://www.tensorflow.org/lite`

on an approximate nearest neighbor that estimate how their distances to the previously-acquired traces. Similar to our approach, it supports various DL domain inputs in its types of mutation, including constrained white noise in signal data like image and audio, and random character perturbations (addition, deletion, and substitution). It also allows finding numerically-unstable responses of the optimized DNN, which can be adapted for erroneous behaviors, and revealing divergences between the original model and its quantized version.

### Evaluation Settings and Metrics

To reduce the adverse effects of both randomness in nature-inspired metaheuristics and selection bias, we experimented on 60 arbitrarily sampled seeds of 32 instances from each included study case's original test data, and all the included estimated metrics are computed as average values over 3 runs or more. For the sake of comparison fairness, we run all the generators with the same number of produced inputs, including metaheuristic-based searcher, random sampler, and TenosorFuzz. Indeed, we use the population/mutation size of 10 and the maximum of iterations of 24, that would yield a total of $10 \times 24 \times 60 \times 32$ generated test inputs. For our approach, we use 6 iterations as a period to eliminate the least relevant transformations and refine the search space regularly. Below, we introduce the different performance metrics that have been used in the evaluations:

**%MI.** It represents the ratio of valid misclassified inputs with respect to the total of all generated synthetic inputs. This enables us to compute failure rates with respect to erroneous DNN behaviors.

**%DI.** It represents the ratio of difference-inducing inputs with respect to the total of all generated synthetic inputs. This allows us to compute failure rates with respect to quantized DNN defects.

**%S[MI/DI].** The percentage of original inputs where at least one misclassification/divergence is successfully revealed among their descendant synthetic inputs. This indicates how diverse the generated failures with respect to their source data

**%Del.** It represents the ratio of test seeds where a selected transformation is deleted during the periodic search space reduction. This helps us identifying transformations that have been recurrently discarded through the test generation.

**RI.** It consists of the ratio of the inertia of transformation vectors generated by a metaheuristic, $I[meta]$ with respect to the inertia of randomly-sampled transformation vectors, $I[RS]$. In general, the inertia of a cloud of datapoints is measured with respect to its center of gravity, G, (i.e., a vector of the mean values), as follows, $I = \sum_{i=1}^{N} \frac{d^2(T_i, G)}{N}$, where $d$ is the Euclidean distance and $N$ is the number of datapoints.

The inertia measures the spread in a cloud of data points, and therefore, our RI measures the dispersion of the transformations generated by our metaheuristics versus the dispersion of a cloud of uniformly sampled transformations.

**Environment**

We developed DeepEvolution in Python, and all the numerical computations are vectorized with the NumPy library. The current version supports TensorFlow (TF) [245] for full-precision DNNs and its TF Lite version for quantized DNNs. All tests were executed in parallel on Google Cloud Platform N2 virtual machines equipped, each with 4 cores and 16 GiB of RAM, and running Ubuntu 18.04. For all preparation of DNNs and tuning of DeepEvolution functions, we use a local server running with Linux CentOS 7 system on Intel(R) Xeon(R) 3104 Bronze with 64 GB of RAM equipped with a NVIDIA GeForce RTX 2080 Ti GPU.

### 6.2.2 Effectiveness of DeepEvolution's Metamorphic Transformations

To assess the utility of the designed semantic-preserving metamorphic relations, we leverage them to conduct average-case analyses of DNN robustness and quantization inefficiency. As an average-case estimation, we use the random sampling algorithm (RS) that draws uniformly compound input distortions from the search space with no guidance based on previous samples. Then, we assess how effective the resulting random metamorphic tests are at revealing faults in DNNs, based on their quantity and diversity. A further analysis is performed on the top-3 transformations by data domains that have been removed during the reduction of the search space.

Table 6.1 reports the results of %MI, %S[MI], %DI, and %S[DI] obtained for DeepEvolution with a random sampling strategy. According to the computed failure (i.e., MI or DI) rates for all the study cases, synthetic distorted inputs can challenge optimized DNNs in robustness against semantically-preserving data transformations, while their quantized versions are also challenged to produce constantly the same predictions as their originals. In light of this observation, even though our designed distortions were intentionally tuned to be within reasonable conditions and the post-transformation validation filter out the too noisy ones, random metamorphic tests uncover hidden weaknesses in the DNNs that the original tests missed because they are biased towards typical and more similar conditions with training samples.

**Distinctions between DL domains and network architectures.** We can also notice from Table 6.1 that metamorphic transformations performed differently across different data

Table 6.1 Performance Metrics of RS-enabled DeepEvolution

| Domain | Model | %MI | %S[MI] | %DI | %S[DI] |
|---|---|---|---|---|---|
| **Image** | EffNET | 23.64 | 99.95 | 11.53 | 99.95 |
| | ViT | 2.54 | 20.52 | 2.33 | 17.97 |
| **Audio** | DepintoCNN | 13.07 | 99.95 | 1.56 | 88.75 |
| | KWS | 6.99 | 88.23 | 14.12 | 97.19 |
| **Text** | BiLSTM | 3.69 | 62.66 | 1.79 | 38.65 |
| | WordCNN | 3.5 | 57.08 | 0.22 | 23.44 |

domains and classification problems. The designed text distortions enhance the diversity of sentences with different vocabulary, but their adverse effects are low since character-level perturbations may lead to insignificant words that the DNN ignores during inference. Indeed, a character-level perturbation less frequently results in removing a crucial word, or producing a new existing word that confuses the DNN. Second, both of our audio and image distortions succeed in yielding high MI rates on DepintoCNN and EffNet. This can be attributed to the relatively-low complexity of the DNNs that have reduced parameter size, which also favor simpler inductive biases and have side effects on robustness. DepintoCNN [227] transforms different-length audio signals into a 40-length feature data based on MFCC, using a single 1D CNN layer to extract the patterns. Despite its effectiveness on the original test dataset, its simple CNN architecture shows weaknesses when faced with naturally-occurring audio signal perturbations. Nevertheless, low-capacity neural networks can be less adversely affected by quantization because the latter will operate on fewer parameters. Accordingly, EffNet [246] is a novel effective CNN architecture that uniformly scales every dimension with a fixed set of scaling coefficients rather than arbitrarily varying width, depth and resolution. In spite of EffNet's high performance on the original CIFAR100 testing dataset, metamorphic tests indicate that this gain in computation complexity and scalability can lead to a lack of robustness when inputs are altered. In contrast, we do not detect a high number of failures for Vit despite Imagenet's high dimensionality. It can be explained by the effectiveness of complex transformers such as Vit, which have been trained with large-size augmented data using different rules that anticipate many of the distortions we include.

**Differences in relevance between domain transformations.** We show in Table 6.2 the deletion ratio of the top-3 transformations that were removed during the search space reduction. For image and audio datasets, the results indicate that the top-3 transformations are responsible for most of the induced variation in the post-transformation validity measurements. Indeed, blur effects, color changes, and brightness changes decrease substantially the SSIM value between the original and transformed images. It is the same for different ad-

ditive noises, including colored, environmental, and random noises, that considerably affect the estimated PSNR of a perturbed audio signal. Meanwhile, the top-3 text transformations have less adversarial effects on the final prediction. In fact, synonyms and embeddings replacement pose less of a challenge in sentiment analysis. Depending on the total number of data transformations included, the deletion ratios of recurrently-discarded transformations vary between domains.

Table 6.2 Deletion Ratio of Metamorphic Transformations

| **Domain** | **Transformation** | **%Del** |
|---|---|---|
| Image | Blur | 33.8 |
| | Color | 27.45 |
| | Brightness | 26.48 |
| Audio | Random Noise | 85.69 |
| | Environmental Noise | 81.30 |
| | Colored Noise | 81.11 |
| Text | Embedding Replacement | 87.5 |
| | Synonym Replacement | 87.45 |
| | Character Removal | 87.08 |

### 6.2.3 Performance of DeepEvolution in DNN Robustness Testing

DeepEvolution is designed to perform worst-case robustness evaluations by leveraging optimization metaheuristics. These latter algorithms drive the sampled distortions towards regions with high fault-discovery capabilities, which likely increases both MI and success rates. In the following, we examine the effectiveness of DeepEvolution in revealing the erroneous behaviors of DNNs. The built-in metaheuristics are compared, as well as DeepEvolution versus TensorFuzz.

Tables 6.3, 6.4, and 6.5 report the performance measures of robustness testing obtained for DeepEvolution with different search strategies, as well as, TensorFuzz Method. In line with expectations, most of metaheuristic algorithms reveal a higher number of MIs, which results in a more substantial MI rates estimated for all classification problems, while preserving and even accentuating the above-discussed differences between them.This is due to RS's absence of logic and to its uniform sampling strategy generating fewer MIs than metaheuristic-based search algorithms, but it will serve us as baseline in the next comparison between the employed metaheuristics. Nevertheless, metaheuristics did not significantly improve the likelihood of converting an original input to a misclassified one, according to the obtained success rates. This illustrates the important role played by the source inputs in deriving synthetic MIs, which is related to the sensitivity of their relevant features to our distortions and/or the

Table 6.3 A comparison of robustness testing performance with different generation techniques for image classification problems

| | **EffNet** | | | | **ViT** | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %AX | %S | RI | Algo | %AX | %S | RI |
| **TensorFuzz** | - | 7.5 | 14.51 | - | - | 0.3 | 4.17 | - |
| **RS** | - | 23.64 | 99.95 | 1.0 | - | 2.54 | 20.52 | 1.0 |
| **Meta#1** | MFO | 72.35 | 99.9 | 1.29 | MFO | 10.45 | 23.33 | 1.56 |
| **Meta#2** | PSO | 52.59 | 99.95 | 1.43 | PSO | 9.85 | 22.97 | 1.41 |
| **Meta#3** | SSA | 32.97 | 99.9 | 3.28 | SSA | 5.62 | 21.09 | 1.12 |
| **Meta#4** | MVO | 27.02 | 99.64 | 1.31 | GA | 5.03 | 23.12 | 1.53 |
| **Meta#5** | GA | 26.76 | 99.95 | 3.77 | WOA | 2.57 | 16.72 | 1.54 |
| **Meta#6** | FFA | 18.61 | 99.38 | 1.61 | MVO | 2.36 | 15.78 | 0.85 |
| **Meta#7** | GWO | 17.11 | 99.95 | 0.51 | FFA | 2.23 | 15.47 | 0.88 |
| **Meta#8** | WOA | 16.44 | 98.65 | 1.66 | GWO | 1.77 | 19.32 | 0.65 |

uncertainty of the DNN against them. The increased exposure of MIs by metaheuristics is of course due to their optimization routines that can reveal a variety of transformations applied to these important original inputs over generations, while RS might find some of them by chance.

**A comparison of the built-in metaheuristics.** We observe that MFO, PSO, and SSA are the top-3 metaheuristics for all the study cases except DepintoCNN, for which they perform well but not better than GWO and FFA. In fact, GWO and FFA often rank low in the other study cases, which indicates a kind of inverse problem. Particularly, all GWO's RI values, including for DepintoCNN, are low, especially for DepintoCNN, which indicates that it favors intensification over diversification. This prevented it from achieving high performance on the MI search problem, and most of the time it was not even able to surpass RS. However, GWO ranks first in testing the robustness of DepintoCNN, meaning that the intensification was effective for its associated MI generation. Further explaining, DepintoCNN is optimized for RAVDESS classification dataset that has different-length audio signals, mostly longer than 1s, while KWS has fixed-length audio signals of 1s. We should consider the maximum length when designing our transformation search space, which increases its dimensionality. In addition, MFCC-based feature engineering transforms varied-length input signals into fixed-length feature data, but this preprocessing may cancel certain distortion effects applied. As a result, the associated search problem is more challenging, and intensification-intensive algorithms can be more effective in uncovering a large number of MIs within previously-discovered faulty regions without wasting time on unnecessary exploration. In the remaining study cases, the top-3 metaheuristics usually have an RI close to 1.0, which means the optimization routines are updating the candidates while maintaining as high a diversity among

Table 6.4 A comparison of robustness testing performance with different generation techniques for audio classification problems

| | **Depinto** | | | | **KWS** | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %AX | %S | RI | Algo | %AX | %S | RI |
| **TensorFuzz** | - | 0.21 | 4.76 | - | - | 0.1 | 4.17 | - |
| **RS** | - | 13.07 | 99.95 | 1.0 | - | 6.99 | 88.23 | 1.0 |
| **Meta#1** | GWO | 28.2 | 100.0 | 0.29 | MFO | 20.96 | 92.76 | 1.04 |
| **Meta#2** | FFA | 25.59 | 100.0 | 1.41 | PSO | 17.13 | 84.27 | 0.94 |
| **Meta#3** | MFO | 25.58 | 99.84 | 1.04 | SSA | 13.37 | 85.68 | 0.9 |
| **Meta#4** | PSO | 22.34 | 99.17 | 0.93 | GWO | 12.27 | 94.11 | 0.2 |
| **Meta#5** | SSA | 21.66 | 100.0 | 0.9 | GA | 11.96 | 89.01 | 1.07 |
| **Meta#6** | GA | 21.23 | 100.0 | 1.17 | MVO | 11.65 | 82.24 | 0.7 |
| **Meta#7** | MVO | 19.02 | 100.0 | 0.78 | WOA | 11.01 | 90.83 | 1.48 |
| **Meta#8** | WOA | 4.11 | 100.0 | 2.01 | FFA | 9.87 | 85.05 | 1.08 |

them as sampling uniformly. Hence, the optimal metaheuristics for DeepEvolution must balance well diversification-intensification in order to continuously improve the generation of test inputs towards increasingly diverse and revealed failures from the defined space. Testing costs can be reduced and the test sessions reflect worst-case robustness assessments, where the derived test inputs stress the weaknesses of the DNN under test. We observe that some metaheuristics fail to surpass the RS performance and most cases have low RI values, which illustrates the importance of diversification to avoid staying too close to previously-obtained successful candidates.

**DeepEvolution vs TensorFuzz.** For all the settings and classifiers, DeepEvolution outperforms TensorFuzz in both terms of MI rates and success rates. The rich data transformations and the search-based approach account for this difference. On one hand, TensorFuzz includes a white-noise mutation for each data type that is continuously applied on the inputs revealing quite distant activations. The resulting successive mutations often lead to invalid and meaningless inputs, discarded by the post-transformation validity check. On the other hand, TensorFuzz selects inputs from the corpus based on their coverage scores without requiring the source data to be diversified, which explains the low success rates. In conclusion, TensorFuzz is designed based on conventional software fuzzing, which makes it more suitable for proving that a DNN can fail, whereas DeepEvolution adapts the search-based approach to reveal as many unique and diverse MIs as possible to estimate the DNN robustness statistically.

Table 6.5 A comparison of robustness testing performance with different generation techniques for text classification problems

| | BILSTM | | | | WordCNN | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %AX | %S | RI | Algo | %AX | %S | RI |
| **TensorFuzz** | - | 1.23 | 4.4 | - | - | 1.61 | 4.82 | - |
| **RS** | - | 3.69 | 62.66 | 1.0 | - | 3.5 | 57.08 | 1.0 |
| **Meta#1** | MFO | 5.28 | 66.93 | 1.0 | PSO | 4.84 | 62.08 | 1.0 |
| **Meta#2** | PSO | 4.7 | 63.65 | 1.01 | MFO | 4.73 | 61.25 | 1.01 |
| **Meta#3** | SSA | 4.34 | 60.05 | 1.01 | SSA | 3.97 | 55.52 | 1.0 |
| **Meta#4** | GA | 3.43 | 50.26 | 1.0 | GA | 3.08 | 44.69 | 1.0 |
| **Meta#5** | WOA | 2.55 | 65.89 | 1.06 | WOA | 2.44 | 63.44 | 1.06 |
| **Meta#6** | MVO | 2.29 | 64.11 | 0.77 | MVO | 2.05 | 62.55 | 0.75 |
| **Meta#7** | FFA | 2.13 | 56.98 | 1.0 | FFA | 1.95 | 56.51 | 0.99 |
| **Meta#8** | GWO | 1.73 | 44.06 | 0.78 | GWO | 1.59 | 41.3 | 0.78 |

### 6.2.4 Performance of DeepEvolution in DNN Quantization Assessment

An additional test objective supported by DeepEvolution is the search for the divergences between an original DNN and its quantized counterpart. The following analyzes the effectiveness of the implementation of metaheuristics in producing relevant distortions that reveal Difference-inducing inputs and, consequently, enhance DI and success rates. We also compare DeepEvolution with TensorFuzz in terms of exposed DIs' quantity and diversity.

Tables 6.6, 6.7, and 6.8 outline the performance measures of quantization assessment obtained for DeepEvolution with different search strategies, as well as, TensorFuzz Method. Similarly to the previous test objective, metaheuristics often reveal DIs higher than RS, and accentuate the difference between two DNNs with different arithmetic precision, if such a gap exists. However, metaheuristics did not help uncover more divergences between the original WordCNN and its quantized version. This can be explained by the low DI rates obtained for all settings indicate that the quantization was successful and there are only a few cases where the two DNNs diverge despite the improved search.

**A comparison of the built-in metaheuristics.** Contrary to the MI search problem, there are no common top-3 metaheuristics that outperform the others in most of the study cases. It is because the DI search problem, involving two similar DNNs under test, challenges the algorithms in discovering the characteristics of fault-revealing transformations. Nevertheless, we noticed that MFO, PSO, and SSA remain constantly the top-3 metaheuristics for the computer-vision classification problems. Additionally, we should emphasize that two metaheuristics, WOA and GA, which ranked previously low, are now competing for first places in the DI generation performance. The correlation between low RI and low failure rates also

Table 6.6 A comparison of quantization assessment performance with different generation techniques for image classification problems

| | **EffNet** | | | | **ViT** | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %DI | %S | RI | Algo | %DI | %S | RI |
| **TensorFuzz** | - | 31.83 | 61.04 | - | - | 0.46 | 7.5 | - |
| **RS** | - | 11.53 | 99.95 | 1.0 | - | 2.33 | 17.97 | 1.0 |
| **Meta#1** | MFO | 32.46 | 99.95 | 1.43 | MFO | 10.74 | 19.9 | 1.28 |
| **Meta#2** | PSO | 20.17 | 99.9 | 1.4 | PSO | 8.07 | 19.38 | 2.22 |
| **Meta#3** | SSA | 15.08 | 99.9 | 2.7 | SSA | 4.99 | 19.48 | 2.69 |
| **Meta#4** | GA | 14.86 | 99.95 | 1.29 | WOA | 3.7 | 19.06 | 2.57 |
| **Meta#5** | WOA | 12.44 | 99.38 | 1.43 | MVO | 3.55 | 17.97 | 0.91 |
| **Meta#6** | MVO | 11.52 | 99.38 | 0.67 | GA | 3.45 | 19.84 | 2.94 |
| **Meta#7** | FFA | 11.2 | 99.9 | 0.87 | FFA | 2.41 | 18.02 | 1.48 |
| **Meta#8** | GWO | 9.51 | 100.0 | 0.83 | GWO | 2.38 | 19.22 | 0.41 |

applies to quantization inefficiency analyses, even though the latter require RIs higher than 1.0 according to the RIs associated with high-ranked metaheuristics. Hence, further exploration capabilities are needed to expose the hidden divergences between two similar DNNs with different arithmetic precision in order to uncover the hidden divergences between them.

**DeepEvolution vs TensorFuzz.** Similar to the previous test objective, DeepEvolution once again outperforms TensorFuzz in both DI rates and success rates. Even when TensorFuzz was able to produce a number of DIs as high as DeepEvolution, its corresponding success rate was low, showing that our approach produces more diverse inputs and is less dependent on the original input. Indeed, TensorFuzz supports the same DL domains, test objectives, and the gray-box nature (exclusive use of the DNN's last layer), as DeepEvolution. Nonetheless, TensorFuzz's white-noise mutations and its fuzzing aspects make it more appropriate for proving the existence of divergences rather than exposing the maximum number of unique and diverse different-inducing instances for further quantization inefficiency analyses.

## 6.3 Threats to Validity

In this section, we address the potential threats to the validity of our research works along with our countermeasures.

**Selection of subjects.** The selection of our experimental subjects, can be a threat to validity. As a mitigation strategy, we use two variants of each evaluation subject with different architectures and all of the considered variants are: (i) established state-of-the-art models, (ii) trained on classification problems widely-used by the community, (iii) pretrained DNNs or training programs from official sources.

Table 6.7 A comparison of quantization assessment performance with different generation techniques for audio classification problems

| | Depinto | | | | KWS | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %DI | %S | RI | Algo | %DI | %S | RI |
| **TensorFuzz** | - | 0.3 | 4.17 | - | - | 0.32 | 4.3 | - |
| **RS** | - | 1.56 | 88.75 | 1.0 | - | 14.12 | 97.19 | 1.0 |
| **Meta#1** | GA | 17.75 | 100.0 | 0.97 | FFA | 31.28 | 93.49 | 1.08 |
| **Meta#2** | FFA | 11.36 | 99.69 | 1.17 | MFO | 22.46 | 92.08 | 1.04 |
| **Meta#3** | SSA | 10.64 | 98.54 | 0.86 | WOA | 21.31 | 97.66 | 1.54 |
| **Meta#4** | MVO | 5.17 | 92.08 | 0.61 | PSO | 20.78 | 87.86 | 0.97 |
| **Meta#5** | MFO | 2.36 | 86.3 | 1.05 | SSA | 19.96 | 92.34 | 0.89 |
| **Meta#6** | PSO | 1.88 | 81.77 | 0.95 | GA | 19.23 | 97.24 | 1.12 |
| **Meta#7** | GWO | 1.69 | 91.35 | 0.24 | MVO | 19.13 | 85.62 | 0.74 |
| **Meta#8** | WOA | 1.51 | 100.0 | 1.5 | GWO | 16.92 | 98.44 | 0.25 |

**Design choices.** The configuration choices that we made throughout the development process can be a threat. To overcome this threat, we implement 8 competitive metaheuristics for the searcher component. We also perform a preliminary assessment of hyperparameters in terms of balance between diversification and intensification. We found that the recommended hyperparameters tested on various benchmark optimization problems are effective for our search problem as well. We observed that further tuning of these hyperparameters compromises the genericity of our approach, increases the costs, and yields dispersed results because the results are not transferable across architectures, classification problems, transformation spaces, and even across different seeds of original test data. Concerning *population_size* and *generation_number* that control the total number of generated samples, we also tried multiple combinations and we ensured that they guarantee fair comparisons, whether it is just the random sampler or TensorFuzz. To ensure the semantic integrity of the generated test inputs, we systematically take random samples of transformed inputs and their associated validity scores to restrict further the ranges of distortion parameters for a higher ratio of valid synthetic data. In addition, we reinforce this validation process with manual inspection (i.e., hearing, seeing, or reading a pair of synthetic input and its original source) by two of the authors on a sample of the synthetic data w.r.t a confidence level of 95% and an error margin of 5%.

**Dealing with randomness.** For reliable conclusions, we conducted experiments that take into account the stochasticity inherent in the nature-inspired metaheuristics. To mitigate the effects of randomness, all of the empirical evaluations of our approach are an aggregative results of independent seed data along with a restart of the process with fresh samples of

Table 6.8 A comparison of quantization assessment performance with different generation techniques for text classification problems

| | **BILSTM** | | | | **WordCNN** | | | |
|---|---|---|---|---|---|---|---|---|
| | Algo | %DI | %S | RI | Algo | %DI | %S | RI |
| **TensorFuzz** | - | 1.04 | 10.42 | - | - | 0.11 | 4.17 | - |
| **RS** | - | 1.79 | 38.65 | 1.0 | - | 0.22 | 23.44 | 1.0 |
| **Meta#1** | WOA | 3.68 | 50.0 | 1.04 | MFO | 0.24 | 18.65 | 1.01 |
| **Meta#2** | MFO | 2.77 | 41.51 | 0.98 | GA | 0.23 | 16.04 | 0.97 |
| **Meta#3** | PSO | 2.72 | 42.6 | 0.99 | PSO | 0.23 | 21.15 | 1.01 |
| **Meta#4** | SSA | 2.58 | 35.89 | 1.0 | SSA | 0.21 | 19.58 | 0.95 |
| **Meta#5** | MVO | 2.19 | 45.89 | 0.78 | GWO | 0.16 | 13.44 | 0.9 |
| **Meta#6** | GA | 1.93 | 29.79 | 1.0 | FFA | 0.13 | 8.91 | 0.75 |
| **Meta#7** | FFA | 1.88 | 37.86 | 0.98 | MVO | 0.12 | 8.75 | 0.57 |
| **Meta#8** | GWO | 1.54 | 28.39 | 0.91 | WOA | 0.07 | 5.52 | 0.58 |

inner random variables.

**Generalizability to other DL applications.** Despite most of related research works on DL testing have focused on CNNs and simple image classification problems such as MNIST mnist and CIFAR10 cifar10, we opt for more complex learning problems across many DL domains that are solved by state-of-the-art DNNs with different optimized neural network architecture. In addition, we detail the common design workflow that we conclude from our experiences to provide a step-by-step process in case there is a need to extend or redefine some components in order to include application-specific transformations or to enlarge the scope of domains. Furthermore, we chose to design and assess DeepEvolution for classification problems because their DL models achieve impressive results and they are widely used by the DL community in several real-world applications across domains and their associated semantically-preserving transformations are relatively straighforward to define, implement and configure based on the label-invariant augmentation rules that have identified and released by the domain experts. Nonetheless, applying DeepEvolution to regression problems shouldn't be difficult as long as we can define the expected output changes respective to the designed input transformation, and hence, a new behavioral drift fitness function should be also designed to capture the deviations induced between the continuous outputs to drive the search in the direction of revealing incorrect ones. Finally, we emphasize that DeepEvolution is implemented in a modular way to support the independent replacement or extension of any functionality.

## 6.4 Chapter Summary

In this chapter, we propose, DeepEvolution, a search-based approach for metamorphic transformation generation as the foundation for a generic-purpose testing framework for DNNs across domains. First, while a test oracle for DNNs are challenging to set up given the cost and laborious tasking of large-scale collection or simulation of novel inputs from scratch, a derived test oracle based on metamorphic relations that are well matched to applied requirements, can provide a good coverage of potential DNN failures in common and edge scenarios. Second, having metamorphic transformations encoded in a vector space that can be systematically explored by metaheuristic searching algorithms, enables the control of test costs and the generation of test inputs directed towards discovering various of target faults: (i) unstable behaviors of optimized DNNs against naturally-occurring input distortions; (ii) divergent behaviors of quantized DNNs compared to their original counterparts. The evaluation is done by using case studies with various recognition models trained on popular image, speech, and natural language datasets. Results show that DeepEvolution successfully exposes the inductive biases' weaknesses and quantization inefficiencies across domains and neural network architectures, and outperforms TensorFuzz, its competitor's coverage-driven fuzzing alternative.

# CHAPTER 7 PHYSICAL: PHYSICS-BASED ADVERSARIAL MACHINE LEARNING

Over the past few years, computational physics and system engineering researchers [247–250] have been increasingly exploring the potential of using machine learning (ML) to alleviate the cost of hand-crafting physical system models. Particularly, deep learning (DL) [251], brought forward a wide array of modern neural network architectures and training techniques. The latter are very effective at approximating any mapping function between variables based on observations and measurements, without prior knowledge of the underlying governing processes. Therefore, the aircraft systems performance models are no exception. Nevertheless, the inexpensive implementation cost of DL-based systems comes with significant trustworthiness concerns, as shown by the adversarial attacks [13]. These issues have become a huge obstacle for DL to overcome in modelling safety-critical systems in civil aircraft product development. First, the experimental data cannot be an appropriate substitute for specifications in the absence of evidence supporting that the data distribution matches the operational conditions. Thus, there is no guarantee that a deep neural network (DNN) trained over a finite set of experimented operational scenarios, could generalize to behave correctly for new operational conditions that were not represented in the original training datasets. Second, the black-box nature of modern learning models and their resulting performance-driven complex architectures no longer allows a full understanding of the complete structural design. Hence, there is no direct method to assess the relevance of the learned latent patterns, whether they support generalizability, or they result from a coincidental optimality of relying on some spurious associations and weak inductive biases.

This chapter presents a novel approach, physics-guided adversarial machine learning (ML), that improves the confidence over the physics consistency of the model. The approach performs, first, a physics-guided adversarial testing phase to search for test inputs revealing behavioral system inconsistencies, while still falling within the range of foreseeable operational conditions. Then, it proceeds with a physics-informed adversarial training to teach the model the system-related physics domain foreknowledge through iteratively reducing the unwanted output deviations on the previously-uncovered counterexamples. Empirical evaluation on two aircraft system performance models shows the effectiveness of our adversarial ML approach in exposing physical inconsistencies of both models and in improving their propensity to be consistent with physics domain knowledge.

**Chapter Overview.** Section 7.1 introduces the essential concepts related to the chapter

content. Section 7.2 presents our novel physics-guided adversarial testing for ML models. Section 7.3 describes a designed physics-informed loss to improve the physics consistency of the on-training ML solution. Section 7.4 reports evaluation results, while Section 7.5 concludes the chapter.

## 7.1 Background

This section describes the essential concepts of adversarial machine learning for regression DNNs, and advances in the physics-guided machine learning.

### 7.1.1 Adversarial Machine Learning for regression DNNs

#### Adversarial Testing for regression DNNs

In the following, we introduce the analytical formulation of the adversarial testing problem. Indeed, the objective of adversarial testing is to design adversary that allows us to detect the data variation ($\delta$) and the natural input ($x$) yielding an adversarial example $\hat{x} = x + \delta$, for which the model does not satisfy a given property $C$. Then, the adversary can be leveraged to improve the satisfiability of the studied property by the model over foreseen inputs. The model robustness represents the first major property that has been relatively well-studied for supervised learning problems using adversarial testing. For regression problems, an epsilon $\epsilon$ is introduced to transform the equality between the resulting discrete outputs into the below-mentioned inequality to support robust comparisons between continuous outputs: $\forall \hat{x}, \quad \|x - \hat{x}\| \leq \delta \longrightarrow |f(x) - f(\hat{x})| \leq \epsilon$.

Robustness against adversarial examples raises the importance of building adversarial attacks and defense techniques aiming at detecting and immuning models against these vulnerabilities early on. While most research studies have focused on local robustness adversarial testing in the context of computer vision system, our present work study the extension of this concept to support other interesting properties: (1) *Model physics consistency* refers to verifying that the predictions are consistent with the physics domain knowledge, which combines first principles and *apriori* system design knowledge; (2) *Model relevance* represents the adequacy of the learning capacity to be less sensitive to overfitting and more robust to fit new samples from the data distribution.

**Adversarial Training for regression DNNs**

Adversarial training is one of the most notable countermeasure, which was first proposed by Goodfellow et al. [71], to improve the robustness of DNNs against adversaries, by re-training the model on the adversarial examples found. For regression problem, Nguyen and Raff [252] add regularization term in the loss function to encourage the numerical stability around delta-local neighborhood. Given $\|\Delta x\|_p < \epsilon$ to denote that we are sampling a point $\Delta x$ uniformly from the $p$-norm ball of radius $\epsilon$, we define adversarial robust loss as follows:

$$l(y, f(x)) + \lambda \times \mathop{\mathbb{E}}_{\Delta x:\|\Delta x\|_p<\epsilon}[l(f(x), f(x + \Delta x)] \tag{7.1}$$

where $\lambda \in \mathbb{R}_+$ controls the strength of defense regularization penalty. In fact, the above-mentioned adversarial loss is composed of two components: the original loss measures the difference between the target output $y$ and the predicted output $\hat{y} = f(x)$, and the regularization penalty imposes that the expectation of the output between a point $x$ and all points within an $L_p$ ball with radius $\epsilon$ around $x$ are the same. Regarding adversarial training, our designed physics-informed adversarial training places more generic cost in the loss function too discourage converging to undesirable input-output mappings, which might be totally incoherent with underlying physics domain foreknowledge.

### 7.1.2 Physics-guided Machine Learning

The impressive efficiency of deep learning in solving industrial problems gave rise to several researches that worked on increasing the performance of data-driven system models using domain knowledge. In [248, 249], a custom loss was proposed based on the domain knowledge, which eliminates entirely the need for supervision data. Karpatne et al. [253] propose to integrate a domain knowledge regularizer in neural networks to influence the model optimization towards better generalization performance. Apart from modifying the loss optimization problem, Ioannou et al. [247] shows that *apriori* structural knowledge could be included into the model architecture design. In a similar fashion, researchers leverage prior knowledge about the problem to incorporate feature invariance [254], to enable representations consistent with physics domain knowledge through either implicit physics rules [255] or explicit physics-based constraints [256]. None of these former researches can be directly applicable to encode the physical relationships required in modelling our target aircraft systems simulators. Instead, our physics-guided adversarial ML approach enables more flexibility as it is designed and implemented independently from both of the deployed DL technologies and the simulated physics-intensive system's details. Indeed, our physics-guided adversarial testing

accepts system-related physics domain knowledge in the format of input-output sensitivity relations with an expected tolerance. Then, it generates optimized test cases to validate, in a black-box fashion, the consistency of the model's input-output mappings that have been statistically inferred from the data. This is also applicable for the proposed physics-informed regularization that performs a data driven fine-tuning of the model relying on the adversarial inputs revealed during the test, which have exposed the model's violations in regards to the specified physics-grounded sensitivity rules.

### 7.1.3 Performance Modeling for Aircraft Systems Simulation

Civil aircraft product development involves complex feedback loops of development and optimization to meet certification and performance requirements. This highly-iterative, complex development workflow requires a combination of extensive domain knowledge, advanced simulation technology, expanded engineering experience and flight test control in order to build high confidence over the designed systems' behaviors. Engineers heavily rely on computerized design aid solutions to model the physical system and assess its conformance with desired requirements regarding principal functionality, safety and reliability. Thus, the system model must be qualified to appropriately reproduce the system's behavior throughout the range of foreseeable operational conditions. This is utterly important to avoid uncovering issues late during production or in-flight testing. Traditionally, physics modelling is used to analyze the system-related physics principles and *apriori* knowledge of system design, in order to develop physics-based models with detailed representations of underlying physics processes and strong priors stemming from first principles and governing equations. Despite the gain in trustworthiness resulting from the structural understanding of physical models' latent variables and equations, they often require comprehensive information about the aircraft physics, and rely on challenging implementation, calibration, verification and validation processes [257, 258]. In the context of aircraft system performance assessment, deep learning technologies allow to quickly infer data-driven models from experimental measurements, with less detailed system knowledge than usually required by physics-based modelling. In particular, we are interested in studying aircraft systems performance simulation models that map stable features at steady-state flight to quantities reflecting the system behaviors, under all the foreseeable conditions in the flight envelope. In fact, time series of aircraft sensor data are collected during flight test campaigns. Then, steady-state flight points are derived and preprocessed meticulously from the collected flight test time series. They represent various flight states in which conditions must remain stable over time for our physical assumptions to hold (i.e. steady-state conditions).Given the high cost of flight tests, aircraft engineers often require such limited-size preprocessed steady-state flight to conduct aircraft subsystems engineering

analysis. Therefore, it is challenging to design and fit a data-driven simulation model that can predict the system behaviors under the operating conditions of interest. For that reason, the data-driven process of DL model development raises trustworthiness concerns about the accuracy of the model on distant data points at test time.

## 7.2 Physics-guided Adversarial Testing

We have developed an approach to assess how well a model is consistent with the foreknown system-related physics domain knowledge. In the following, we detail the three principles of the proposed approach, its integrated components, and the resulting physics-guided adversarial testing workflow.

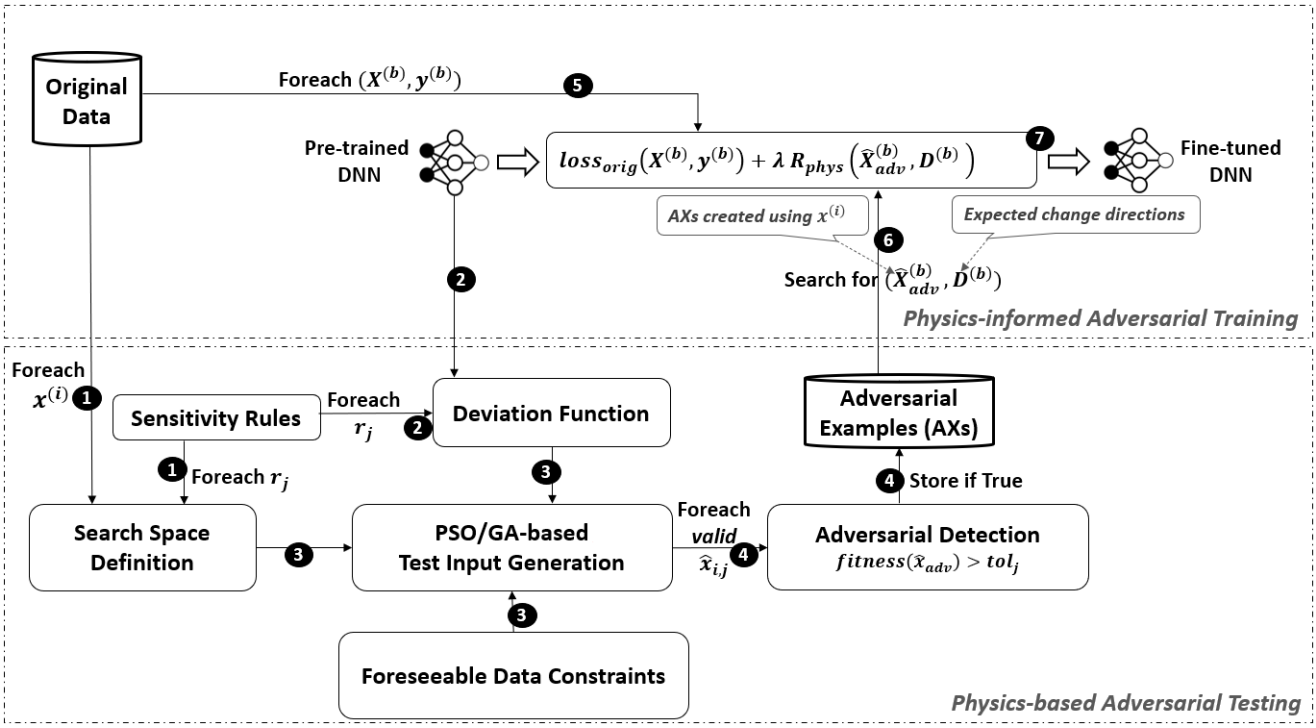



Figure 7.1 Overview of Physics-Guided Adversarial Machine Learning Phases and Workflow

### 7.2.1 Specification of Physics Domain Knowledge

Aircraft product development heavily relies on modelling physical processes and phenomena to enable the interpretation of the interactions between the input quantities, $x$, and dependent variables, $y$ that are observed during a flight. In contrast, data-driven modelling consists

of inferring statistically the mapping between the input quantities and the observed variables from the collected experimental data without prior knowledge of the underlying governing physics phenomena. Both system modelling approaches would serve similar engineering use cases that mainly require the assessment of system behavior at different operating conditions (i.e., normal and extreme flight scenarios). Even if the model's mapping function does not include explicitly physics equations, the statistically-inferred mappings should respect the system-related physics processes as well. Hence, we propose the specification of the system's desired properties in terms of must-hold relationships between the input sensors data and the target quantity of interest. Mathematically, the system domain expert codify these foreknown relationships in a format of sensitivity rules that map a subset of oriented input features variations (i.e., signed value changes) to anticipated variation trend of the output. For instance, we might foreknow that increase of $x_1 \nearrow$ and/or decrease of $x_2 \searrow$ lead to the increase of $f(x_1, x_2) = y \nearrow$. These sensitivity rules are designed to be applied locally over the genuine data points when system engineering experts can confirm that combined input features variations usually map to higher order effects that are not necessarily detectable locally within predefined bounds. Furthermore, we highlight that our defintion of sensitivities include the input features variations leading to no trend conclusion (i.e., a constant model output). These special mappings with invariant output are commonly used to capture physics invariance principles, where the dependent quantity remains unchanged under certain circumstances. In the following step, we deep dive into how the local search space is defined to keep the transformed test inputs in the neighbourhood of the original data point and constantly meet the foreseeable data conditions.

### 7.2.2 Inference of Physics-Guided Adversarial Tests

In this DL adversarial testing, we broaden the scope of adversarial tests to cover not only invariance properties such as imperceptible image perturbations (largely applied to computer-vision models), but also to assert the model's input-output sensitivities with physics-grounded expectations. Given the specified sensitivity rules, we can infer two main types of adversarial tests depending on the expected deviation of the predicted quantity under the conditions on input features:

**Invariance Test:** It restricts the input generation to the datapoint-wise local neighbours that should share equal predicted quantity, in order to verify the consistency of the model with experts' invariant-output mappings, reflecting physics invariance principles. The test assertion consists of the equality test between the model's predictions on the original and its derived synthetic data points.

**Directional Expectation Test:** It applies the constrained perturbation of input features, as outlined in the premises of the specified sensitivity rules. The resulting synthetic neighbours data should lead to the expected directional deviation on the model's predictions, as foreknown by the experts in the conclusion of the sensitivity rules. Thus, the test assertion depends on inferring the outputs of the model for both original datapoint and its craftly-perturbed neighbors, then, makes sure the directional expected deviations happen (i.e., the synthetic datapoints' predictions should be either higher or lower than the original datapoint's prediction).

Although the defined adversarial tests rely on sensitivity rules grounded by theoretical physics laws and processes, they would be applied on experimental sensors data that might encouter different sources of noise. Hence, we decide to soften the equations and inequalities involved in the above-mentioned tests assertions by including a tolerable deviation error, $tol_i$, on any measured sensor input, $x_i$, according to domain expert guidances.

### 7.2.3 Search-based Approach for Physics-Guided Adversarial Testing

The backbone of the proposed physics-guided adversarial testing is the metaheuristic-based optimization that searches for the erroneous behaviors of the DNN against the specified physics-grounded sensitivity rules. Below, we explain the development steps that we follow to construct this search-based testing approach.

#### Definition of the data search space

A physics-guided adversarial test may be one of two types depending on the assertion. It is either to assert if the output is invariant, or if the expected directional deviation occurs. The physics-based sensitivity rules specify the assertion type in their conclusions. As a result, the input search space can be defined similarly for both types of physics-guided adversarial tests. From the premises of the underlying sensitivity rule, we derive the space boundaries, which include the upper and lower bounds of each involved input feature. For instance, a sensitivity rule, stating that $x_1 \nearrow$ and $x_2 \searrow \longrightarrow y \nearrow$, corresponds to the subspace of all the inputs, $\hat{x}$, where $\hat{x}_1 > x_1$ and $\hat{x}_2 < x_2$. For each inequality in the feature space, however, we still do not have the other side boundary. We could complete the missing boundaries in the above-mentioned inequalities of feature space by using the full range of expectations, $[m_i, M_i]$ for each input feature $x_i$. Hence, the search space of inputs in the example above is $\hat{x}$, where $x_1 < \hat{x}_1 < M_1$ and $m_2 < \hat{x}_2 < x_2$. Other input features not mentioned in the rule's premises are denoted, $x_c$, and they are supposed to remain constant. In order to account for the experimental noise, we set their values range to be between

$[min(m_i, x_i - tol_c), max(M_i, x_i + tol_c)]$. The search of the derived input space corresponds to sampling the neighbors of the original datapoints, for which the specified sensitivity rule should permanently apply. As a result, these sampled neighbors datapoints would serve as physics-based adversarial test inputs.

**Definition of the foreseeable data constraints**

Any part of an aircraft must meet its intended performance requirements in all foreseeable operating conditions in order to be certified. United States Code of Federal Regulations (CFR), part 25, for Jet aircraft design states that: **"the equipment, systems, and installations must be designed and installed to ensure they perform their intended functions under all foreseeable operating conditions."** To simulate aircraft components under different operating conditions of interest, the designed performance models should include input features that reflect flight conditions (e.g., altitude, speed, and outside temperature), in addition to the system-related sensor data. As a result, our test cases should pertain to aircraft flight envelopes and foreseeable ambient conditions in order to represent meaningful and useful simulation scenarios. There are physics processes that govern the interactions between ambient conditions measurements and aircraft operating envelopes, which specify, as an example, the maximum airspeed the aircraft can reach at a given pressure altitude. Therefore, we define data constraints limiting the input search space to the foreseeable ambient and flight conditions. In order to be considered as valid test cases, the sampled inputs must satisfy the defined foreseeable data constraints. In our performance models of aircraft systems, we derived the foreseeable pairs of input features (altitude, airspeed) and (altitude, ambient temperature) from the flight envelopes. Additionally, the operating airspeed is further delimited when high-lift devices are extended (e.g., slats and flaps), according to the aircraft operating conditions. To increase the likelihood of generating inputs that satisfy all of the foreseeable data constraints, the latter are also incorporated into the input generation problem.

**Design of the fitness function**

The fitness function, $fitness(\hat{x})$, should directly measure how much the model's prediction for the input $\hat{x}$ is inconsistent with the specified physics sensitivity rule, $r_j$. Thus, the generated input data with higher fitness values would have high chances to fail the underlying adversarial test. Given the model $f$ under test, an original input $x$ and its derived synthetic $\hat{x}$ resulting from applying the rule, $r_j$, the deviation from the desired behavior would depend on the type of the adversarial test. Regarding the invariance test, we verify constant-output

rules $r_j \in R_{cons}$ where the absolute difference between original and synthetic predictions, $|f(x)-f(\hat{x})|$, can be the behavioral deviation. Concerning the directional expectation test, we validate either increasing-output rules $r_j \in R_{incr}$ or decreasing-output rules $r_j \in R_{decr}$, where the behavioral deviation would be, respectively, $f(x)-f(\hat{x})$ and $f(\hat{x})-f(x)$. Hence, the fitness function should be equal to the behavioral deviation measure, $fitness(\hat{x}) = dev_{f,r_j}(x, \hat{x})$ that can be formulated as follows:

$$dev_{f,r_j}(x, \hat{x}) = \begin{cases} f(x) - f(\hat{x}), & \text{if } r_j \in R_{incr} \\ |f(x) - f(\hat{x})|, & \text{if } r_j \in R_{cons} \\ f(\hat{x}) - f(x), & \text{if } r_j \in R_{decr} \end{cases} \tag{7.2}$$

Using the behavioral deviation measure $dev_{f,r_j}(x, \hat{x})$, we can generalize the assertion of both adversarial test types to be the inequality, $dev_{f,r_j}(x, \hat{x}) > tol_j$, where $tol_j$ is the tolerance predefined for the output quantity.

**Implementation of metaheuristic-based optimizers**

We apply SBST, using population-based metaheuristics to drive optimally the data generation towards diverse prominent regions in the input space of the underlying adversarial test. In line with the No Free Lunch Theorem (NFL) [235], we implement two concurrent population-based metaheuristics, PSO and GA, that are described in 2.2.1. Then, we tune each metaheuristic algorithm's hyperparameters to appropriately tune its level of nondeterminism and balance between the intensification (exploiting the results and concentrating the search on regions near effective solutions found) and diversification (exploring non-visited regions to avoid missing interesting potential solutions) [242]. In our test data search, we aim for the optimal balance between new test inputs that are sufficiently different from the old ones to uncover new regions of interest while at the same time similar enough to test inputs that have high fitness values to uncover more adversarial examples, which belong to the interesting regions that were previously found. Besides, the enhancement of the fitness values, over iterations, produces new test inputs with higher deviations with respect to the expected behavior, and hence, these test inputs would have likely better chances to expose DNN's physics inconsistencies. There is, however, no optimal or suboptimal solution that would represent, in our case, the test input with the highest fitness. Consequently, we modify slightly the standard design of population-based metaheuristics to continuously monitor adversarial examples among evolving feasible solutions. A feasible solution is any valid test input that successfully meets all the sets of foreseeable constraints. A valid test input that violates the underlying physics sensitivity rule constitutes an adversarial example.

By completing all the above construction phases, we arrive at the proposed physics-guided adversarial testing workflow shown in Figure 7.1, which lays out the steps in the following order: *(1)* For each original input, $x^{(i)}$, and sensitivity rule $r_j$, the search space is instantiated according to the premises of $r_j$ and the features' values of $x^{(i)}$; *(2)* With the current sensitivity rule $r_j$ and the model under test $f$, the deviation function, $dev_{f,r_j}$, would be as in Eq. 7.2; *(3)* The population-based metaheuristic algorithm would search over the input space for the most-fitted entries, i.e., the ones triggering high deviation values and satisfying all the foreseeable constraints; *(4)* For all the generated synthetic inputs $\hat{X}_{i,j}$, a follow-up adversarial test would assert if the computed deviation, $dev_{f,r_j}(x_i, \hat{x}_{i,j})$, exceeded a prefixed threshold, which by default equals the rule tolerance. $tol_j$. Each revealed adversarial example, $\hat{x}_{adv}$, should also be stored along with its metadata (parent index $i$, expected deviation $d$, permissible tolerance $tol_j$).

As the leveraged metaheuristic algorithms are iterative, our workflow encapsulates a nested loop of the steps, *(3)* - *(4)*, that would be repeated for a total of maximum iterations, $K$. Indeed, the searcher starts at the first iteration, $k = 0$, with a population of candidate inputs, $\hat{X}_{i,j}^{(k)}$, randomly sampled from the neighbors of $x^{(i)}$ w.r.t the premises of $r_j$. Then, it computes their fitness scores, as follows, $\textit{fitness}(\hat{x}) = dev_{f,r_j}(x^{(i)}, \hat{x})\ \forall \hat{x} \in \hat{X}_{i,j}^{(k)}$, and captures all the $\hat{x}_{adv}$ that meet the condition of $\textit{fitness}(\hat{x}_{adv}) > tol_j$. Afterwards, the metaheuristic update routines evolve the population, $\hat{X}_{i,j}^{(k)}$, and derive new candidates, $\hat{X}_{i,j}^{(k+1)}$, that are probably better than their predecessors in terms of fitness. Thus, the main loop of our workflow consists of running all of the steps ($1 \rightarrow 4$) for all the original inputs, $X$, as well as for all the applicable sensitivity rules $R$. Thus, the semi-supervised adversarial examples, $\hat{X}_{adv}$, would serve us later in fine-tuning the DNN's mappings to capture the underlying physics processes and system-related properties.

### 7.3 Physics-informed Adversarial Training

Regularization penalties are often developed by ML scientists that reflect model complexity and encourage optimizers to find simpler mapping between features and outputs. Thus, the loss function would be written as follows, $l(y, \hat{y}) + \lambda.R(\theta)$. The widely-used regularization penalties are $L_1$-norm and $L_2$-norm on the parameters, which respectively impose low-magnitude parameters and sparse parameters (i.e., with zeroed coefficients). In this section, we propose a physics-informed regularization by devising a penalty that encourages the training algorithm to maintain a reasonable level of physics consistency in the learned mapping function.

### 7.3.1 Physics-informed Regularization Cost

A follow-up fine-tuning procedure will leverage revealed adversarial examples to fix these erroneous behaviors, similarly to conventional adversarial learning 7.1.1. Nevertheless, it is natural to retrain using the semantically-preserving adversarial examples as we expect no change in class label or slight deviation from continuous output. In contrast, we need to be prepared for a deviation in the predicted outputs in response to our produced synthetic inputs, $\hat{x}$, but we are aware of the correct direction of the change in the model's predictions, based on the physics-based sensitivity rule, $r_j$. As a result, we introduce a variable $d$ equals to 1 or $-1$ if we expect the output to increase ($r_j \in R_{incr}$) or decrease ($r_j \in R_{decr}$), respectively, and $d = 0$ if we do not expect any significant change in the output ($r_j \in R_{cons}$). Therefore, we aim to design a physics-informed regularization that uses the semi-supervised, generated test data assembling the transformed inputs $\hat{x}$, the expected change direction $d$, and the reference pointed to its parent input $x$. As a first step, we were inspired by the *hinge_loss*= $(0, 1-f(x).y)$, which is commonly used for maximum-margin classifiers [259], such as support vector machines (SVMs). The hinge loss is a specific type of cost function that incorporates a margin from the hyperplane, representing the classification decision boundary. Thus, it penalizes even correctly-classified data points if they are very close to the hyperplane, i.e., their distances are less than the margin. In addition to placing the data points on the correct side of the hyperplane, the hinge loss encourages the classifier to place them beyond the margin as well. Indeed, the distance from the hyperplane can be regarded as a measure of confidence. Therefore, we estimate the physics-consistency error to reflect how far the model deviates from the expected change direction, using the deviation function (Eq. 7.2). Then, we ignore any noise-induced deviations within the margin of error by using the domain-specific tolerance, *tol*, defined by experts. Afterwards, we square the non-zero deviations to give a stronger weighting to larger differences and to follow the same scale as the squared prediction errors. The resulting regularization cost for the model $f$ under test can be formulated in the below Eq. 7.3.

$$R_{phys}(x, x_{adv}) = \begin{cases} (max(tol, f(x) - f(\hat{x})) - tol)^2, & \text{if } d = 1 \\ max(tol^2, (f(x) - f(\hat{x}))^2) - tol^2, & \text{if } d = 0 \\ (max(tol, f(\hat{x}) - f(x)) - tol)^2, & \text{if } d = -1 \end{cases} \tag{7.3}$$

where $x_{adv} = (\hat{x}, d, tol)$, $x, \hat{x} \in \mathbb{R}^D$, $d \in \{-1, 0, 1\}$, $tol \in \mathbb{R}_+$.
Then, we generalize the regularization cost in a single generic function that includes all the types of our physics-based adversarial test assertions, as below formulated in Eq. 7.4.

$$R_{phys}(x, x_{adv}) = [max(tol^{p_2}, ((-1)^{p_3}(f(x) - f(\hat{x})))^{p_2}) - tol^{p_2}]^{p_1} \tag{7.4}$$

where $p_1 = 1 + d^2$, $p_2 = 2 - d^2$, $p_3 = |d|(\frac{d+1}{2} + 1)$.
Furthermore, we would emphasize the importance of including a regularization tolerance, *tol* as a predefined hyperparameter, which can be equal, by default, to the tolerance used in adversarial detection to overcome experimental noises. In the absence of this tolerance, the physics-informed regularization would likely push the model towards having, empirically, zero cost in terms of violating theoretical physics rules, but based on experimental data. Hence, we run the risk of stagnation and perhaps even divergence as we attempt to strictly apply the physics rules to noisy data.

### 7.3.2 Physics-informed Adversarial Training Algorithm

For the sake of simplicity, we have presented our physics-informed regularization cost estimation on a pair of original, $x$, and adversarial inputs, $\hat{x}$. To avoid sub-optimal local minima, we usually use mini-batch stochastic optimization for training DNNs. So, we will also apply a mean reduction strategy to the regularization costs of all inputs in a batch. The mean over the sum reduction is chosen because it preserves the invariance of the loss scale to the batch size, and it is aligned well with the mean squared error, commonly used as data loss for regression models. Therefore, the physics-informed loss function that sums the original loss and the cost of its integrated regularization for batches of data ($X^b$, $y^b$) can be defined as follows:

$$l_{phys}(X^b, y^b, X^b_{adv}) = l(f(X^b), y^b) + \lambda_{phys} R_{phys}(X^b, X^b_{adv})$$

where $X^b_{adv} = (\hat{X}^b, d, tol)$, $x \in \mathbb{R}^{B\times d}$, $\hat{X}^b \in \mathbb{R}^{M\times d}$, $d \in \mathbb{R}^M$, $tol \in \mathbb{R}^M_+$, $\lambda_{phys} \in \mathbb{R}_+$ controls the strength of the physics-informed regularization penalty.
In our preliminary experiments, we observed that the choice of lambda, $\lambda$, could be challenging and hinder the performance of physics-informed adversarial training. Below we describe the two identified major challenges in lambda, $\lambda$, setup with respect to the substantial differences in magnitude of both costs (namely, data loss and regularization).
**Unfair cold start conditions between both costs:** In fine-tuning, we start with pre-trained DNN and its corresponding adversarial examples that highlight its revealed inconsistencies with the underlying physics sensitivity rules. As a consequence, it is natural to have initially low data loss, $l(f(X^{(b)}), y^{(b)})$, and a high physics-informed regularization cost, $R_{phys}(X^{(b)}, X^{(b)}_{adv})$. Thus, the fine-tuning process focuses primarily on minimizing physics-informed regularization cost, while being excessively tolerant of substantial increases in data loss, as long as the total sum-up loss, including the regularization penalty, continues to decrease. As a solution to this starting bias, we set lambda, $\lambda$, in a way that aligns the magnitudes of both costs from the beginning (i.e., the first iteration) in order to start from

fair initial conditions.

**Substantially different sizes of adversarial batches over iterations:** Despite the use of mean reduction in each cost to normalize over the batch data, we face cases where the original batch has no corresponding adversarial examples or may have several of them. This induces substantial differences between the batch data loss, $l(f(X^b), y^b)$, and its related physics-informed penalty cost, $R_{phys}(X^b, X^b_{adv})$; so we dynamically update the lambda value, $\lambda_{phys}$, in order to adapt the magnitude of the additional penalty cost depending on its initial order of magnitude, whenever it exists (i.e., not null). As a reference loss value, we use the average loss estimated over all the batches using the best fitted model before starting the fine-tuning, because the batch losses could substantially differ too and destabilize the convergence.

To illustrate how the physics-informed adversarial training algorithm works, we describe the remaining steps of the workflow in Figure 7.1 and we refer to the lines of code from the Algorithm's Pseudo Code 7.1: *(5)* The algorithm guarantees non-divergence from the best-fitted state as it iterates over batches of original data, $X^{(b)}$, (code lines 6-7) and watches the model's loss on them (code line 13), which means there is no substantial degradation in regards to the fit of the original distribution; *(6)* It searches all AXs, $\hat{X}^{(b)}_{adv}$, and their meta-data including expected change directions $D^{(b)}$ and deviation tolerance $T^{(b)}$, which have been revealed relying on the original input, $X^{(b)}$ (code line 10), and computes their corresponding physics-informed regularization cost, $R_{phys}(X^b, X^b_{adv})$ (code line 15); *(7)* It dynamically calibrates the magnitudes of both losses (code line 17), by inferring the lambda coefficient, $\lambda$, (code lines 22-26) to keep the regularization cost aligned with the base loss (i.e., the model's average data loss at launch).

```
#prepare the base data loss as the model loss before fine-tuning
preds = DNN(X)
base_data_loss = MSE(y, preds)
for epoch in epochs:
   indices, X, y = shuffle(X, y)
   batches = data_loader(indices, X, y, batch_size)
   for indices_b, X_b, y_b in batches.iterate():
        preds_b = DNN(X_b)
        #search all the adversarial examples connected to the current batch
            entries
        X_adv, d_adv, t_adv = adversarial_data.search(indices_b)
        preds_adv = DNN(X_adv)
        #compute the data loss
        data_loss = MSE(y_b, preds_b)
```

```
        #compute the physics regularizatio cost
        reg_cost = R_phys(preds_b, preds_adv, d_adv, t_adv)
        #compute dynamically lamda to align the magnitudes of both losses
        lamda = pow(10, magnitude_order(base_data_loss/reg_cost))
        #aggregate both losses
        loss = data_loss + lamda * data_loss
       DNN = update_model(DNN, loss)

def magnitude_order(value):
    '''computes the magnitude order of a real value.
       For example, (0.004) --> (-3), (105) --> (2)
    '''
    return math.floor(math.log(value, 10))
```

Pseudo-Code 7.1 Physics-Based Adversarial Training Algorithm

## 7.4 Evaluation

In this section, we introduce two industrial case studies, as well as our evaluation setup, metrics, and methodology. Next, we test the effectiveness of our physics-guided adversarial machine learning approach for assessing and improving the trustworthiness of neural networks used to simulate aircraft performance.

### 7.4.1 Experimental Setup

#### Case Studies

Two aircraft systems performance simulation models were used in our empirical evaluation. The first case study consists of an aircraft performance (referred to as *A/C Perf.*) model mapping steady-state angle of attack ($\alpha$) to features related to flight conditions and wing configurations. The second case study consists of an in-flight wing anti-icing performance (known as *WAI Perf.*) model mapping wing leading-edge skin temperature sensors to features related to flight conditions, wing configurations, and high-pressure pneumatic system conditions at the wing root. Indeed, Jet aircraft often use hot-air ice protection systems that prevent ice accumulation over the wings during flight. Thus, the performance of the In-flight wing anti-icing system is determined by its ability to sustain, under all foreseeable conditions, a wing leading-edge skin temperature sufficient to prevent ice formation. These case studies were run on the regression data sets summarized in the Table 7.1 where $N$ is the number of

steady state flights and $D$ is the number of dimensions. The problem-related data (features

Table 7.1 Size and dimensionality of data sets

| Dataset | N | D |
|---|---|---|
| Aircraft Performance Data | 1338 | 5 |
| Wing Anti-Icing Performance Data | 90 | 10 |

and targets) as illustrated in Table 7.2 and 7.3, were obtained from flight test campaigns and were provided as a dataframe, with entries representing distinct steady-state flights, and columns representing selected features. Indeed, these steady-state flight points are derived

Table 7.2 Aircraft performance Data Catalog

| | **Name** | **Type** |
|---|---|---|
| **Features** | Calibrated Airspeed | Real |
| | Aircraft Weight | Real |
| | Slat Angle | Dichotomous |
| | Flap Angle | Categorial |
| **Target** | Angle of Attack | Real |

Table 7.3 WAI Performance Data Catalog

| | **Name** | **Type** |
|---|---|---|
| **Features** | Angle of Attack | Real |
| | True Airspeed | Real |
| | Pressure Altitude | Real |
| | Pressure of Wing Root | Real |
| | Temperature of Wing Root | Real |
| | Total Air Temperature | Real |
| | Slat Angle | Dichotomous |
| | Flap Angle | Categorical |
| **Target** | A-Wing's Skin Temperature Sensor | Real |
| | B-Wing's Skin Temperature Sensor | Real |

and preprocessed meticulously from costly flight test time series. They represent various flight states in which conditions must remain stable over time for our physical assumptions to hold (i.e. steady-state conditions). As can be observed, the number of flight datapoints may differ substantially due to the difficulty of relevant steady state extraction. In our study cases, the A/C performance model lies on standard flight parameters (i.e., speed, altitude, and weight), which are often found to be stable during flight. Hence, we were able to collect numerous datapoints that correspond to conditions commonly encountered in flight test

campaigns. On the other hand, the steady-state datapoints for WAIS performance modeling are quite difficult to extract. They require the aircraft pilot to fly for quite a long time (i.e, up to tens of minutes) during which the WAI system is precisely configured. Given the high cost of flight tests, aircraft engineers often require such limited-size preprocessed steady-state flight to conduct aircraft subsystems engineering analysis. They employ extensive advanced analytics combined with their domain knowledge to provide the necessary demonstrations for certification with the minimal data points. This is known as critical point analysis (CPA). For example, certification of engine icing protection mechanisms [260] requires the analysis of the most critical points that provide evidence of the systems' effectiveness across the entire ice envelope under all foreseeable operating conditions (hold, descent, approach, climb, and cruise). With either a limited set of critical points or a high coverage of normal operating conditions, it is challenging to design and fit a data-driven simulation model that can predict system behaviors under all the foreseeable conditions in the flight envelope. To circumvent the limited size of datasets, the engineering team defined system-centric transformations, as outlined in Table IV. Given the limited range of variation $t$ within [-10, 10], the one-to-many augmentation rule, $A_1$, mimics different equilibrium states of temperatures based on the genuine flights to enhance the diversity among the training samples. The one-to-one augmentation rules, $A_2$, $A_3$ and $A_4$, serve as boundary conditions that help inject artificial data points with the aim of improving the model's data fitness. In the following, we describe the augmentation rules from Table 7.4 and their rationales. The expected variation of

Table 7.4 Augmentation Rules for WAIS Perf. Model

| **Status** | **Rule** | **Premises** | **Conclusion** |
|---|---|---|---|
| ON | $A_1$ | TAT + $t$ , TWR + $t$ , $\forall t \in [-10, 10]$ | $T^{a,b}_{skin} + t$ |
| | $A_2$ | PWR = 0 | $T^{a,b}_{skin}$ = TAT |
| | $A_3$ | TAS = 0 | $T^{a,b}_{skin}$ = TWR |
| OFF | $A_3$ | TWR = TAT | $T^{a,b}_{skin}$ = TAT |

leading-edge skin temperatures, $T^{i}_{skin}$, $i \in a, b$, must be approximately equal to the variations of temperature at wing (TWR) and the total air temperature (TAT) when the later is equal and within a predefined range. In $A_1$, aircraft engineers have fixed an absolute difference less than 10. In $A_2$, the rule sets the PWR to 0, eliminating the flow of WAI through the wing, which consequently cancels the effect of TWR on the skin temperatures. Thus, they become equal to TAT. In $A_3$, the rule simulates synthetic data points at extreme conditions, by at-

tributing 0 to the airspeed (TAS). Theoretically, this results in no forced convection and the wing skin cannot transfer heat to the outside air. Thus, both $T^a_{skin}$ and $T^b_{skin}$ should be equal to TWR. In addition, the augmentation rule, $A_4$ can be applied when WAI is deactivated. It sets TWR to be equal to TAT; so TAT and TWR are set to the same value. Accordingly, the two leading-edge skin temperatures must be approximately equal to this value. For further details on the heating and cooling principles governing the WAI system, we refer to our explanation of its physics-grounded sensitivity w.r.t each input quantity in Section 7.4.1. Thereby, the training data loader incorporates an online augmentation step that randomly transforms the mini-batches feeded to the on-training model. It helps prevent overfitting as the model rarely encounters the exact same inputs multiple times and cannot simply memorize them. Our online augmentation randomly applies one of the defined augmentations half of the time, i.e., a genuine training input can be transformed with a probability of 0.5. As certain simulation-driven risk analyses are conducted under extreme and rare conditions with low coverage, our physics-based adversarial ML leverages both system-related and physics domain knowledge to validate and improve the physics consistency of the statistically-learned models; this makes them more viable alternatives to purely physics-based simulators. In addition, the flight envelopes for the studied Jet aircrafts, which included the parameters of interest, were gathered from the internal specifications of Bombardier aircraft products.

**Physics-based Sensitivity Rules**

In the following, we briefly expose the physic-based high-level requirements for the above-mentioned systems in terms of input-output sensitivity rules using expert know-how. The specifications of the rules are inferred from a detailed knowledge of the aircraft performance principles [261] and the local heat transfer characteristics [262] that arise from energy, momentum and mass balance over an aircraft wing.

**A/C Performance Model.** The performance model of the studied aircraft is associated to the fundamental performance characteristics of an aircraft. It associates the aircraft orientation to the amount of weight it can lift at a certain speed. The model uses calibrated airspeed (CAS), a quantity that is by definition strongly correlated to the true airspeed. The model also assumes non-accelerated flight at small flight path angles; hence the lift is approximately equal to the aircraft weight (ACWT). From the laws of motion and basic knowledge of air flow around thin bodies, one can infer several sensitivity relations between system's inputs and output. First, one can state from Newton's third law that aircraft wings create an upwards force by deflecting air flow downwards. When airspeed increases, less deflection is necessary to produce the same force. This relation holds true for thin, smooth profiles at low deflection angles, for which the airflow deflection is assumed to be strongly correlated to

the angle of attack ($\alpha$). Thus, $\alpha$ is inversely correlated to CAS at constant ACWT. Second, Newton's third law also justifies the assumption that greater air flow deflection generates more lift, allowing to carry more weight. Therefore, $\alpha$ is positively correlated to ACWT at constant CAS. Third, aircraft's slats and flaps are devices designed to increase lift when flying at low speed. Hence, it is expected to observe operationally a lower aircraft $\alpha$ as these high-lift devices are deployed. Therefore, in normal operations, $\alpha$ is expected to be negatively correlated to the slat deployment bit, and to the flap angle. These three a priori observations are then encoded into physics-based sensitivity rules, as shown in Table 7.5.

**WAI Performance Model.** During flight, the anti-icing capability of jet aircraft wings

Table 7.5 Physics-grounded Sensitivity Rules for A/C Perf. Model

| Rule | Premises | Conclusion |
|---|---|---|
| $r_0 \in R_{incr}$ | CAS $\downarrow$ , ACWT $\uparrow$ , Flap $\downarrow$ , Slat $\downarrow$ | $\alpha \uparrow$ |
| $r_1 \in R_{decr}$ | CAS $\uparrow$, ACWT $\downarrow$, Flap $\uparrow$, Slat$\uparrow$ | $\alpha \downarrow$ |
| $r_2 \in R_{cons}$ | CAS $\leftrightarrow$ , ACWT $\leftrightarrow$ | $\alpha \leftrightarrow$ |

is determined by the control of the leading-edge skin temperature, measured as $T^i_{skin}$, $i \in a, b$, which prevents ice from forming on the wings. For hot-air anti-icing systems, adequate skin temperatures are assured by the hot air flow stream entering the wing at temperature (TWR) that compensate for the loss of temperature caused by the ambient air surrounding the wing (at temperature TAT). As a matter of fact, an increase of TAT and TWR will result in warmer skin temperatures. Based on energy conservation and convection heat transfer [263], the efficiency of the energy exchange between the internal and external air streams can be encoded with the following sensitivity rules with focus on some key variables. First, an increase of WAI pressure at the wing root (PWR) induces increased internal air flow to raise the skin temperatures, $T^a_{skin}$ and $T^b_{skin}$. Then, a higher aircraft air speed (TAS) eases the heat exchange with the outside air, which pushes skin temperatures closer to those of the outside air stream. Inversely, an elevated altitude (ALT) causes the air density to decrease along with the heat exchange with the outside, which consequently raises both of $T^a_{skin}$ and $T^b_{skin}$. Also, these skin temperatures are negatively correlated to the angle of attack ($\alpha$) which affects the pressure and airflow temperature distributions above the wings, causing fluid to accelerate more rapidly in the upper side wing areas. When the WAIS is turned off, the effects of input features, namely, TAS, ALT, and TWR become negligible, which results in more compact sensitivity rules with less variables in the premises. A wing skin temperature, $T^i_{skin}$, $i \in a, b$, reaches the total air temperature (TAT) at steady-state if WAIS is deactivated. Hence, varying the aircraft's speed or altitude does not directly affect the skin temperatures, $T^a_{skin}$ and $T^b_{skin}$, but they influence the atmospheric conditions outside the aircraft. We already account

for these effects with the concept of total air temperature (TAT). Furthermore, they are not affected by variations in TWR, since there is no flow of WAI passing through the wing. Last, the slat extension (SLAT) and flap angle (FLAP) were discarded in the sensitivity rules for the wing anti-icing system (WAIS) because their effects on skin temperatures are difficult to characterize. SLAT and FLAP variations induce local changes in the flow behavior over the wing surface, but the resulting sensitivity of $T^i_{skin}$, $i \in a, b$ depends on the exact location of the wing temperature sensor. Characterizing the sensitivity would require advanced modeling based on a precise analysis of the local flow patterns with computational fluid mechanics modeling methods. Therefore, Table 7.6 summarizes all the physics-based sensitivity rules that can be exploited by our approach on WAI performance model's during a physics-guided test session.

Both system models lack physics-grounded invariant rules, but we added a constant-output

Table 7.6 Physics-grounded Sensitivity Rules for WAIS Perf. Model

| **Status** | **Rule** | **Premises** | **Conclusion** |
|---|---|---|---|
| ON | $r_3 \in R_{incr}$ | ALT $\uparrow$ , TWR $\uparrow$ , PWR $\uparrow$ , TAT $\uparrow$ , TAS $\downarrow$ , $\alpha \downarrow$ | $T^{a,b}_{skin} \uparrow$ |
| | $r_4 \in R_{decr}$ | ALT $\downarrow$ , TWR $\downarrow$ , PWR $\downarrow$ , TAT $\downarrow$ , TAS $\uparrow$ , $\alpha \uparrow$ | $T^{a,b}_{skin} \downarrow$ |
| OFF | $r_5 \in R_{incr}$ | TAT $\uparrow$ , PWR $\uparrow$ , $\alpha \downarrow$ | $T^{a,b}_{skin} \uparrow$ |
| | $r_6 \in R_{decr}$ | TAT $\downarrow$ , PWR $\downarrow$ , $\alpha \uparrow$ | $T^{a,b}_{skin} \downarrow$ |
| Both | $r_7 \in R_{cons}$ | ALT $\leftrightarrow$ , TWR $\leftrightarrow$ , PWR $\leftrightarrow$ , TAT $\leftrightarrow$ , TAS $\leftrightarrow$ , $\alpha \leftrightarrow$ | $T^{a,b}_{skin} \leftrightarrow$ |

sensitivity rule for each model, $r_2$ and $r_7$, that describes the degree to which the model's output can change as input features change. Following expert guidance, domain-specific tolerances are used to define the radius of perturbations on the inputs and the maximum unsigned deviations of the outputs. Indeed, the invariance tests are essential to ensure the smoothness of the learned mapping function as well as its numerical stability to be a viable simulation solution for one of these smooth or piecewise-smooth dynamical systems.

### Models

Our base nonlinear regression model is a feedforward neural network(FNN) that is trained using the Mean Squared Error (MSE) loss function with L2-norm regularization. Rectified

linear units (ReLU) are used as hidden layer activation functions, and Adam is used as the optimization algorithm. In regards to the architecture, we followed the design principle of pyramidal neural structure [136], i.e., from low-dimensional to high-dimensional feature spaces/layers, as well as these dimensions are powers of 2 to achieve better performance on GPUs [264]. Regarding the preprocessing, both of data inputs and outputs are scaled to have zero mean and unit standard deviation before getting used by the training algorithm in order to standardize their units.

**A/C Performance FNN.** It assembles three layers with, respectively, 128, 64, and 32 neurons. The best-fitted parameters were obtained by a training of 250-epochs using a batch size of 64, a learning rate of $1e-4$ and an L2-norm weight decay coefficient of $5e-4$.

**Wing-anti Icing Performance FNN.** It stacks the consecutive layers including 256, 128, and 64 neurons. The best-fitted state was reached with 300 epochs of training using a batch size of 16, a learning rate of $1e-3$ and an L2-norm weight decay coefficient of $1e-4$.

Next, we describe the hyperparameters tuning strategy we adopt to find the above FNNs for our case studies.

### Hyperparameters

For the FNNs hyperparameters tuning, we leverage the random search strategy to sample several trials of the settings, and we use out-of-sample bootstrap validation that enables stable estimations for relatively small datasets, in order to infer the expected predictive performance of each configuration. A model is trained using a bootstrap sample (i.e., a sample that is randomly drawn with replacement from a dataset) and tested using the rows that do not appear in the bootstrap sample. Indeed, we use a 100-repeated out-of-sample bootstrap process, where the resampling with replacement is repeated 100 times. To outline the ranges of the different tuned hyperparameters, we denote $linspace(a, b, n)$ to indicate the range of $n$ equi-spaced values within $[a, b]$ and $logspace(c, d, base)$ to indicate the interval of $base^c, base^c + 1, .., base^d$, where $c < d$. Starting with the capacity-related hyperparameters, the depth of the neural network is selected from $linspace(1, 6, 1)$, and the size of layers is sampled from the binary logarithmic space, formulated as $logspace(5, 10, 2)$, which keep the enabled learning capacity for the FNNs in accordance with the actual complexity of our case studies. Then, the remaining hyperparameters were tuned as follows: learning rate $\eta \in s \cup 3 \times s$, weight decay $\lambda \in s \cup 5 \times s$, where $s = logspace(1, 5, 10)$. Batch size was tuned in $logspace(3, 7, 2)$, and epochs count in $linspace(50, 500, 50)$. Concerning the hyperparameters that control the exploration-exploitation trade-off of the involved nature-inspired population-based metaheuristics, we opt for grid search strategy to assess their performances under different alternative settings and find the best configuration in terms of

the total count of the revealed unique valid inputs. For PSO, $w$, $\varphi_p$, and $varphi_g$ were tuned in $linspace(0.1, 0.95, 0.05)$, which lead us to set up PSO-enabled test generator with $w = 0.8$, $\varphi_p = 0.65$, and $varphi_g = 0.75$. For GA, $p_{mutation}$ and $r_{parents}$ were tuned, respectively, in $linspace(0.1, 0.95, 0.05)$ and $linspace(0.1, 0.5, 0.05)$, which ends up with the configuration of $p_{mutation} = 0.25$ and $r_{parents} = 0.3$. In addition, several binary crossovers for breeding were tested such as one-point [59], two-point [59], or uniform [59], and the one-point crossover operation outperforms the others for our test input generation problem. Last but not least, we rely on system engineering experts' judgment and the operating specifications of the aircrafts used in the flight test data in order to fix the tolerances and the confidence intervals in relation with the features and targets in our case studies.

**Evaluation Strategy and Metrics**

Due to the limited-size of flight test data for both studied system performance, we adopt a 10-fold cross-validation method for all experiments in order to have different splits for the training and validation datasets and quantify the target metrics by averaging their values over the 10 iterations. Besides, all the included estimated metrics are computed as average values over 5 runs or more, in order to mitigate the effects of randomness inherent in meta-heuristic search algorithms. Below, we introduce the different evaluation metrics that have been used in the empirical evaluations.

**%ValIn.** It represents the ratio of valid inputs with respect to the total of all generated inputs. Valid inputs are those that comply with all the foreseeable constraints that encode nonlinear interactions between the input features, as determined by the aircraft operating conditions and atmospheric conditions.

**%DupIn.** It represents the ratio of duplicate inputs with respect to the total of all adversarial inputs. In our semi-supervised adversarial datasets, duplicate entries are defined by a pairwise Euclidean distance, i.e., two real-valued input vectors with a distance of zero are considered to be duplicates.

**#AdvIn.** It consists of the number of adversarial inputs, i.e., those that contradict our physics-based sensitivity rules based on their corresponding deviation function, as formulated in Eq. 7.2.

The following are two metrics based on Percentage Change, which represents the degree of change relative to a base starting point. It can be the percentage of either increase or decrease, which are basically the amount of, respectively, increase or decrease, from the original quantity to the final one in terms of 100 original parts.

**%Improv_AdvIn.** The improvement we achieve by regularizing can be estimated using the percentage of decrease in the revealed adversarial test inputs from the original model to

its regularized counterpart, as formulated in the below Eq. 7.5.

$$\%Improv\_AdvIn = \frac{pre\text{-}\#AdvIn - post\text{-}\#AdvIn}{pre\text{-}\#AdvIn} \times 100 \tag{7.5}$$

Where *pre-#AdvIn* and *post-#AdvIn* refer to the actual number inconsistencies (#AdvIn) revealed before, i.e., using the original version of FNN and after the physics-informed adversarial learning, i.e., the regularized version of FNN. **RMSE.** It stands for Root Mean Square Error, which averages all the quadratic deviations between the predicted values and the true/observed ones, and then computes its square root to have an error measured on the same scale as the output. Deviations are proxies for expected prediction errors, since the estimations are based on an out-of-sample, unseen test dataset. In general, RMSE is non-negative and lower values are better than higher ones.

**%Change_RMSE.** It consists of the percentage increase, as formulated in the Eq. 7.6, because it is unknown in advance how physics-informed regularization will affect prediction errors. Thereby, positive values (highlighted in red) indicate the on-watch metric, RMSE, increase whereas negative values (shown in green) indicate RMSE decrease.

$$\%Change\_RMSE = \frac{post\text{-}RMSE - pre\text{-}RMSE}{pre\text{-}RMSE} \times 100 \tag{7.6}$$

To obtain domain experts' feedback, we recruited two senior engineers from our industrial partner, Bombardier Aerospace: The first engineer works in the aircraft performance team and provides us with feedback regarding the first study case. The second engineer is a specialist in the thermodynamics engineering team and has the expertise required for the second study case. Afterwards, we interviewed them separately to gather their opinions on the strengths/weaknesses of the proposed physics-guided adversarial machine learning, and their experience on the usage of the web-based interface and its provided configurable parameters.

### Software

The physics-guided adversarial learning framework was developed in Python. It uses Pytorch [265], an established DL framework for modelling and training neural networks. In addition, our implemented search-based techniques were based on an adaptation of the open-source python libraries, pyswarm[1] and geneticalgorithm[2], to meet the specifications of our designed population-based metaheuristic search algorithms. As a user-friendly version for

[1] https://pythonhosted.org/pyswarm/

[2] https://pypi.org/project/geneticalgorithm

the collection of domain experts' feedback, we develop a web-based interface for the physics-guided adversarial ML approach in order to facilitate the live user tests.

### 7.4.2 Experimental Resuls

We assess the effectiveness of the proposed approach using the following research questions:

***RQ1. How effective is the approach at detecting physics inconsistencies?***

**Motivation.** The objective is to assess how effective our approach is at testing the physics consistency of a DNN by exposing adversarial examples that highlight problematic regions in the input space, where the model deviates from the foreknown domain knowledge.

**Method.** We experiment our testing approach using different sizes of data generations, i.e., total of generations $G$ equals to the product of generations count, maximum of iterations, the number of rules, and the number of original inputs, while counting and storing the found adversarial examples. Then, we run the physics-guided adversarial testing on both performance models at each iteration of the cross-validation process, using training and validation dataset splits. Thus, all the estimated counts would be an average of the obtained adversarial examples over the separate cross-validation splits. Besides, we turn off our metaheuristic-based searching algorithm by switching to randomly sampling inputs from the search space of the underlying sensitivity rule, and verifying their validity against the foreseeable data constraints. This input random sampler (RS) represents our simple and inexpensive baseline to assess the added value of the metaheuristic-based searching algorithms in improving the fitness of test inputs over the course of generation. In order to ensure a fair comparison, we sampled a total of random inputs equal to the previously-calculated size of data generations for the studied nature-inspired metaheuristics, PSO and GA.

**Results.** Table 7.7 shows the occurrences counts of unique adversarial inputs (i.e., their predictions are inconsistent with physics-based sensitivity rules) revealed by each search-based method for each performance model, each dataset split, as well as increasing trial sizes.

As can be seen, the average counts of unique adversarial examples (i.e., Column Avg. #AdvIn in Table 7.7) are non-zeros. Actually, the foreseeable synthetic inputs that our approach produces to stress the conformity of models with physics-grounded sensitivity rules, have exposed physics inconsistencies regardless of the employed search algorithm.

**Finding 1:** The designed physics-based adversarial testing successfully reveals the neural network's inconsistencies against domain knowledge, assembling physics first principles

Table 7.7 The number of unique exposed adversarial cases on train and test datasets per search method and system

| SYS | Method | G | Dataset | Avg. #AdvIn |
|---|---|---|---|---|
| A-C Perf. | RS | 361K | $D_{train}$ | 1302.3 |
| | | 40K | $D_{test}$ | 128.1 |
| | | 903K | $D_{train}$ | 3186.4 |
| | | 100K | $D_{test}$ | 340.3 |
| | PSO | 361K | $D_{train}$ | 9340.3 |
| | | 40K | $D_{test}$ | 1127.4 |
| | | 903K | $D_{train}$ | 25134.4 |
| | | 100K | $D_{test}$ | 2827.7 |
| | GA | 361K | $D_{train}$ | 919.7 |
| | | 40K | $D_{test}$ | 75.6 |
| | | 903K | $D_{train}$ | 2161.0 |
| | | 100K | $D_{test}$ | 209.4 |
| WAI Perf. | RS | 361K | $D_{train}$ | 29.5 |
| | | 40K | $D_{test}$ | 3.3 |
| | | 903K | $D_{train}$ | 67.2 |
| | | 100K | $D_{test}$ | 9.4 |
| | PSO | 361K | $D_{train}$ | 1099.8 |
| | | 40K | $D_{test}$ | 113.2 |
| | | 903K | $D_{train}$ | 3884.6 |
| | | 100K | $D_{test}$ | 400.4 |
| | GA | 361K | $D_{train}$ | 17.0 |
| | | 40K | $D_{test}$ | 1.2 |
| | | 903K | $D_{train}$ | 56.1 |
| | | 100K | $D_{test}$ | 12.1 |

and *apriori* system design knowledge.

A comparison between the used search algorithms gives us the following ordered sequence: PSO, RS, then GA. The PSO-enabled generator succeeds in revealing the highest number of physics inconsistencies, and unexpectedly, the GA-enabled generator fails to even outperform the baseline, RS. To further compare the behaviors of these algorithms, we compute the average ratio of valid inputs generated by each algorithm, along with the average ratio of duplicate inputs that are discarded during the generation process, as shown in Table 7.8. Indeed, the complexity of the encoded conditions could cause the search algorithm to stagnate in invalid regions without generating enough foreseeable inputs to test the model. In addition, a non-optimal exploitation/exploration trade off configuration would also cause the search algorithm to loop over the same regions of previously-triggered adversarial inputs, resulting

in a high duplicate rate.

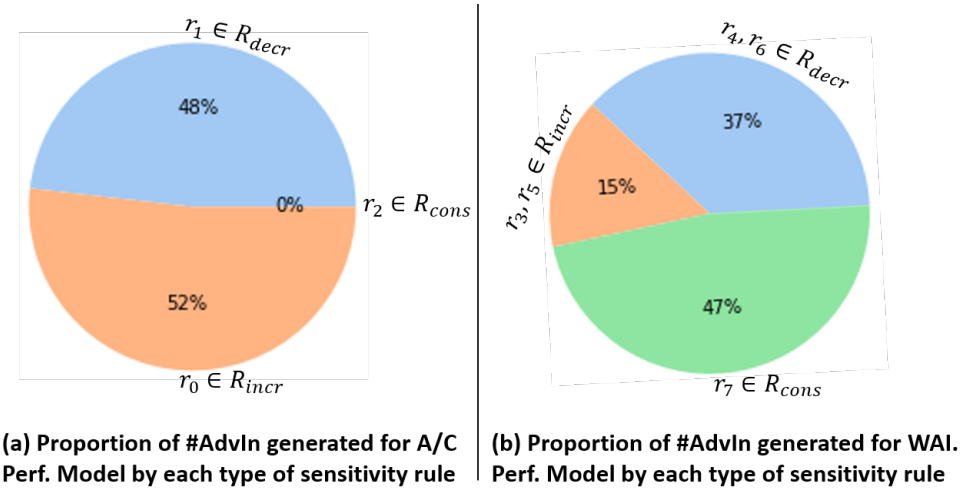
Table 7.8 Average ratio of valid test inputs and duplicated adversarial examples per SBST algorithm and per system

| SYS | ALG | Avg. %ValIn | Avg. %DupIn |
|---|---|---|---|
| A-C Perf. | RS | 31.40% | 0% |
| | PSO | 44.35% | 31.61% |
| | GA | 27.55% | 10.47% |
| WAI Perf. | RS | 90.16% | 0% |
| | PSO | 99.12% | 81.56% |
| | GA | 82.45% | 34.84% |

As can be seen in Table7.8, the average ratio of valid inputs per searching algorithm suggests the same sequence order we had in the comparison with respect to the counts of adversarial inputs. It is expected that the search approach that is able to remain in valid input regions longer, is more likely to find more adversarial examples. Although PSO was the most successful method in terms of valid and failed inputs, it also produced the highest ratio of duplicate inputs. We find this to be a strong indicator of over-exploitation, as PSO might have been heavily relying on the most-fitted previous solutions, which kept it rolling over the same solutions indefinitely. These observed differences in the search behaviors of the evaluated algorithms indicate that they produce inputs of varying diversity. Hence in the following research questions, we consider all three search algorithms in assessing the regularization based on physics domain knowledge, both in terms of fixing the physics inconsistencies and improving the prediction errors. We intend to explore the effects of AXs source generators on model improvements.

**Finding 2:** our PSO-enabled testing approach outperforms GA-enabled version and RS baseline in regards to the number of successfully-exposed physics inconsistencies of the trained neural networks.

Considering the smoothness of the simulated systems' dynamics, a weight decay regularization was applied to system performance models to promote the learning of smooth mapping functions and penalize unnecessarily large response changes. Nevertheless, Figure 7.2 shows the proportion of the adversarial inputs generated by our approach using each type of sensitivity rule for (a) the A/C performance model, which is resilient to invariance tests, and (b) the WAI performance model, which is highly sensitive to input perturbations with a large proportion of failed invariance tests. This observed difference in proneness to injected input noises can be attributed to the complexity of the system and the size of the datasets. The




**(a) Proportion of #AdvIn generated for A/C Perf. Model by each type of sensitivity rule**

**(b) Proportion of #AdvIn generated for WAI. Perf. Model by each type of sensitivity rule**

Figure 7.2 Proportion of AdvIn generated by each Type of Sensitivity Rule

limited number of steady-state flights with the underlying WAI system deployed prevents the NN-based simulator from learning numerically stable states, leaving it susceptible to invariant adversarial examples. Thereby, our physics-informed adversarial training relies on failed invariance tests to increase the numerical stability of the neural network over subsequent fine-tuning steps. Moreover, Figure 7.2 shows that the A/C performance model (a) produces almost equally-distributed directional expectation tests, precisely the increasing and decreasing output rules. Alternatively, the WAI performance model (b) generates a higher percentage of AXs based on decreasing output rules as compared to the increasing output counterparts. This observed difference can be explained by the proximity to normal behaviors. On the one hand, the WAI system should maintain the skin at warm or hot temperatures as necessary, depending on the ambient conditions. On the other hand, the decreasing-output sensitivity rules define the input perturbations causing the outputs to decrease. Thereby, these sensitivity rules derive synthetic flight datapoints, having skin temperatures that are expected to be lower than the original flight test inputs. Hence, the resulting decreased-output datapoints are capable of simulating rare and extreme operating conditions to challenge the original performance model trained on left-skewed data distribution (i.e., skin temperatures tend to be relatively high compared to the foreseeable range).

**Domain Experts Feedback.** Aircraft development engineers perceive the potential value of our proposed adversarial testing approach in the ML-based simulator development. They affirm that the sensitivity analysis represents a main step in the system modelling to validate their expectations through what-if scenarios on how the target variables should be affected based on changes in the input variables. The what-if question would be like “what would

happen to the quantity of interest $q$ if the input variable $var_1$ went up by 1%". Therefore, they consider our proposed SBST approach as kind of a large-scale automation for searching over all the possible what-if scenarios and reporting the ones triggering unanticipated behaviors. Nonetheless, our system engineering stakeholders emphasize the challenges of defining the application scope of such physics-grounded sensitivity rules in regards to more sophisticated systems. Indeed, further restrictions on the local subspace of perturbed inputs would likely take place when combined variations of system inputs as well as relatively high magnitude changes provoke non-negligible higher order effects on the target output.

***RQ2. How well does the physics-informed regularization fix the physics inconsistencies?***

**Motivation.** The goal is to assess the usefulness of our proposed physics-informed regularization in immuning the DNN against physics-based adversarial examples. In other words, we want to assess how well the regularized model learns the underlying physics sensitivity rules.

**Method.** The adversarial examples detected by each search algorithm during the second test session, when a larger number of generations is involved, are loaded onto the training datasets. Then, we perform physics-informed adversarial training that combines the conventional data loss with our proposed regularization cost for the adversarial examples revealed in the training datasets. Thus, we re-run the adversarial detection process on the adversarial examples exposed for its corresponding validation dataset. We aim to check whether the detected physics inconsistencies in the validation data stay or disappear. This allows us to assess the effectiveness of our physics-informed adversarial training phase in fine-tuning the neural network to the foreknown relationships between the input and output variables. As discussed in the previous research question, we keep all three search algorithms in the analysis to compare their resulting regularization improvements, separately.

**Results.** Table 7.9 compares the number of physics inconsistencies (#AdvIn) revealed before and after the physics-informed adversarial learning, as well as the estimated improvement ratios (%Improv_AdvIn) for each supported search algorithm.

As illustrated by the improvement ratios in Table 7.9, the regularized neural network has triggered less physics inconsistencies in regards to the validation datasets for both of the studied performance models and the three implemented search algorithms. Therefore, the inclusion of physics-informed regularization costs improves the neural networks' propensity to be consistent with the underlying physics sensitivity rules.

Besides, search algorithms with higher counts of exposed AXs for the base neural network

Table 7.9 comparison between #adversarials before and after fine-tuning fix

| **SYS** | **G** | **pre-#AdvIn** | **post-#AdvIn** | **%Improv_AdvIn** |
|---|---|---|---|---|
| A-C Perf. | RS | 5267 | 1012 | 80.78% |
| | PSO | 39551 | 5747 | 85.46% |
| | GA | 2850 | 636 | 77.68% |
| WAI Perf. | RS | 509 | 0 | 100% |
| | PSO | 20545 | 18 | 99.91% |
| | GA | 459 | 4 | 99.12% |

have achieved higher improvement ratios (%Improv_AdvIn) of the physics inconsistencies. Indeed, sorting the search algorithms by the column of Table 7.9, #pre-AdvIn (numbers of exposed AXs for base NNs), returns PSO, RS, followed by GA, which achieved on average, respectively, 92.69%, 90.4%, and 88.4%. The ranks of search algorithms obtained by %Improv_AdvIn are in agreement with their ranks by #AdvIn (i.e., number of exposed AXs) on both training and testing datasets, as shown in Table 7.7. Given the statistical grounds and data-driven nature of the proposed physics-informed regularization cost, this result is expected. In fact, our physics-guided adversarial training incorporates soft penalty terms for violations of physics-based sensitivities to the loss function. Then, it proceeds with data-driven repairs of the neural network using the semi-supervised AX datasets detected on the training datasets. Thus, the search algorithms that yield larger AX datasets during the adversarial testing, provide more physics inconsistencies to teach the model the specified physics-grounded sensitivities over fine-tuning steps with the proposed composite loss function. Therefore, their corresponding regularized neural networks are unlikely to behave inconsistently with the foreknown sensitivities, even on the synthetic inputs crafted from test datasets that might differ from the ones built using the training examples.

In addition to the influence of the AX data size, we also noticed that the initial problem size measured by the dimensions of the original data, affects the effectiveness of our physics-informed adversarial training. In Table 7.9, we can observe that the improvement ratios obtained for the WAI performance model are in-between 99%-100%, in comparison with the achieved improvements on the A/C performance model, ranging from 77% to 85%. This distance in the number of fixed physics inconsistencies between the two studied simulation problems could be explained by the absolute number of revealed AXs and the size of the original datasets, as demonstrated in Table 7.1. As can be seen, the WAI performance data has less original datapoints (90) and more feature dimensions (10), which results in a low input space coverage and less diversity in the revealed AXs, while the A/C performance data has more datapoints (1334) and less feature dimensions (5), yielding a higher input space

coverage and more diversity in the revealed AXs.

**Finding 3:** The physics-informed regularization improves the quality of the learned model's mappings towards more consistency with the physics domain knowledge, but the achieved improvements depend on the source of physics-based AXs and the genuine input space coverage.

**Domain Experts Feedback.** Regarding the data-driven corrections of the ML model's physics consistency, system engineering experts introduce us the possible noises in the test flight sensors data that are leveraged to train and test the ML models. In fact, earlier versions of our approach did not include an expert-defined tolerable deviation ($tol_j$) that has been used for both of the follow-up adversarial input test and physics-informed regularization cost. Formerly, our designed adversarial ML approach reports consistently a high number of revealed adversarial inputs due to a slightly deviation of their outputs in the undesired direction. Moreover, the rigid physics-informed regularization cost without tolerance parameter was kind of pushing the fine-tuned neural model towards over-respecting the designed theory-grounded rules on the sensors data collected during monitored flight tests.

***RQ3. Does the physics-informed loss improves or degrades the performance of the DNN?***

**Motivation.** The purpose of this research question is to determine whether physics-informed loss affects the performance of the fine-tuned models indirectly.

**Method.** We took the fine-tuned models from the physics-informed adversarial training, and we computed their predictive performance, i.e., the root mean squared error (RMSE), on the original data.

**Results.** The RMSE and #Change_RMSE values, shown in Table 7.10, are obtained from the evaluation of neural networks on the validation datasets after and before performing the fine-tuning sessions on semi-supervised datasets that include the adversarial examples found on the training examples.

Based on Table 7.10, we found that random sampler (RS) with no searching capabilities caused the RMSE to either stall or increase in all the assessment experiences. In contrast, GA-enabled input generation leads to only negative %Change_RMSEs, which means that the post-fix RMSEs have been successfully decreased. PSO, which was the most effective in exposing the physics-based AXs with higher #AdvIn, often degrades post-regularization RMSEs (i.e., yielding positive #Change_RMSE) at the cost of reducing the inconsistencies with respect to the physics-grounded sensitivity rules. As demonstrated by the study case

Table 7.10 Comparison between RMSE after and before fine-tuning fix

| SYS | TRG | pre-RMSE | ALG | post-RMSE | %Chg RMSE |
|---|---|---|---|---|---|
| A-C Perf. | $\alpha$ | 0.498° | RS | 0.497° | -0.20 |
| | | | PSO | 0.996° | +100 |
| | | | GA | 0.444° | -10.84 |
| WAI Perf. | $T^a_{skin}$ | 4.088°C | RS | 4.729°C | +15.68 |
| | | | PSO | 4.422°C | +8.17 |
| | | | GA | 3.979°C | -2.67 |
| | $T^b_{skin}$ | 7.524°C | RS | 7.921°C | +5.28 |
| | | | PSO | 6.826°C | -9.28 |
| | | | GA | 7.163°C | -4.80 |

on A/C performance, PSO found much higher AXs than other competitors (see Table 7.7) and achieved better physics consistency improvements (see Table 7.9). The post-fix RMSE, however, was doubled, with %Change_RMSE equal to 100%.

In our experimentation, GA-enabled adversarial testing has identified few but useful adversarial inputs that improve post-regularization RMSEs, contrary to PSO and RS, which were less effective in identifying adversarial examples that could positively affect the predictive performance of regularized neural networks after fine-tuning. While aircraft performance studies often involve limited-size datasets of expensive flight tests, they are intended to train simulation models of the system's expected behavior under all foreseeable operating conditions. Hence, an increase in the estimated errors on validation data will be acceptable within a certain system-dependent range validated by domain experts to ensure that the statistically-derived mappings of the model are reliable and fairly consistent with the physics domain knowledge. Using the same A/C performance model as an example, our collaborators contend that a regularized model with a 1° of average error and high consistency with the specified physics sensitivities over the flight envelope can be more useful in different engineering applications than an A/C performance model with low physics consistency and a 0.5° of average error.

**Finding 4:** A physics-informed adversarial training's effect on the predictive performance heavily depends on the size and quality of the revealed adversarial input data. According to our experiments, these AX criteria can be controlled by selecting the appropriate search algorithm. Specifically, the GA-enabled generator reveals a few physics-based AXs that improve the prediction, while the PSO counterpart reveals many AXs, but at the risk of degrading the original predictions.

**Domain Experts Feedback.** Regarding the probable degradation induced by the physics-informed adversarial training, aircraft system engineering experts highlight the importance that data-driven models statistically infer input-output mappings, which are thoroughly consistent with the physics-grounded sensitivity rules. Thus, they suggest that the physics-guided adversarial approach should allow the user to manually balance out the tradeoff between the data loss and the physics-informed loss during the physics-guided adversarial training, depending on the system modelling use case. In response to that, we design a weight parameter, $\beta$, to control the regularization cost similarly to the traditional norm penalties, while keeping the dynamic calibration of $\lambda$ that we added to overcome the differences in the losses magnitudes. Thus, we will have $\lambda = \beta \times \lambda_{dynamic}$, where the $\lambda_{dynamic}$ is the actual pre-computed lambda value and the $\beta \geq 1$ is the weight of regularization cost, by default equals to 1. For some use cases, system engineers can set up higher $\beta$ values to prioritize the model's physics consistency within the input space over further improvements on the RMSE.

## 7.5 Chapter Summary

The present chapter proposes a physics-guided adversarial machine learning approach that assembles: (i) a physics-guided adversarial testing method that successfully exposes physics inconsistencies in ML-based A/C systems performance models; (ii) a physics-informed adversarial training approach that promotes learning input-output mappings, satisfying the desired level of consistency with physics domain knowledge. A physics inconsistency in the input space could be expected owing to the complexity of the simulated system dynamics and the rarity of the flight test data. Our physics-based adversarial testing applies search algorithms to conduct worst-case analysis on the model's inconsistencies and provide insights on their prevalence in the input space. Our subsequent physics-informed regularization always improves the physics consistency of the model, but we observed that a high density of exposed AXs might degrade its predictive performance.

# CHAPTER 8 SMOOD: SMOOTHNESS-BASED OUT-OF-DISTRIBUTION DETECTOR

Aircraft industry is constantly striving for more efficient design optimization methods in terms of human efforts, computation time, and resources consumption. Hybrid surrogate optimization maintains high results quality while providing rapid design assessments when both the surrogate model and the switch mechanism for eventually transitioning to the High-Fidelity(HF) model are calibrated properly. Due to its accurate predictions and fast inferences, feedforward learning networks(FNNs) can effectively address the imperfect surrogate modeling issue faced by conventional machine learning models in the field of aircraft design optimization [266]. However, FNNs often fail to generalize over the out-of-distribution (OOD) samples, which hinders their adoption in critical aircraft design optimization. The resulting corner-case behaviors adversely affect the overall performance assessments for the design configurations, which in turn increases the aircraft design lead time and development budget. However, chasing complete training sets that cover all the facets of the distributions in relation to the quantities of interest, restarts the onset limitation of compute-intensive costs, since the data in aircraft design is typically generated from expensive numerical simulations. Furthermore, the construction of advanced surrogate models using state-of-the-art deep learning techniques [266–268] requires model engineering efforts, draws on new expertise, and often results in over-optimized models that target individually specific sub-problems.

In response to the aformentioned challenges, we propose *SmOOD*, a smoothness-based OOD detection approach, that allows to codesign the surrogate neural network model for accurate assessments and its OOD detector for selective prediction. Unlike common research on out-of-distribution, *SmOOD* exploits the apriori smoothness property of the simulated system to overcome the main challenges of OOD detection, including complex, high-dimensional input spaces and degeneration of uncertainty estimates beyond the ID regions. By using *SmOOD*, only high-risk OOD inputs are forwarded to the HF model for re-evaluation, leading to more accurate results at a low overhead cost. Three aircraft performance models are investigated. Results show that FNN-based surrogates outperform their Gaussian Process counterparts in terms of predictive performance. Moreover, *SmOOD* does cover averagely 85% of actual OODs on all the study cases. When *SmOOD* plus FNN surrogates are deployed in hybrid surrogate optimization settings, they result in a decrease error rate of 34.65% and a computational speed up rate of 58.36×, respectively.

**Chapter Overview.** Section 8.1 provides background information about surrogate modeling for aircraft design. Section 8.2 presents our novel smoothness-based OOD detection approach for NN-based surrogate models. Section 8.3 reports evaluation results, while Section 8.4 summarizes the chapter.

## 8.1 Surrogate Modeling for Aircraft Design

Surrogate models are data-driven and low-cost substitutes for the exact evaluation of the data points in design space, sometimes dominating the entire optimization process [269], and sometimes serving just as a supplementary aid to speed up the computations [270, 271]. Multiple techniques have been proposed in the literature to build data-fit surrogates that are trained with regression of high-fidelity simulation data. They usually rely on methods such as gaussian processes(GP) [272–274], proper orthogonal decomposition [275], eigenvalue decomposition [276], artificial neural network [277, 278], and more advanced techniques such as combining GP and neural networks [279]. In the context of aircraft design optimization, the selection of the suitable surrogate modeling technique depends on the complexity of the underlying design problem, the ease of collecting high-fidelity simulation data, and the cost of development and maintenance. Surrogate modeling solutions are usually presented in conjunction with specific MDO problems from different industries. Hence, advanced techniques pose challenges in regards to the engineering efforts required to adapt them. This chapter examines established data-driven modeling methods such as GP and FNN that can be applied directly to a wide variety of MDO problems, including aircraft design optimization. We developed a variety of surrogate models for different quantities of interest (QoI) using mainstream DL frameworks and GP modeling tools. These QoIs capture certain performance factors of an aircraft and are dependent on the operating flight conditions and the design variables. Our focus is on their deployments to accelerate the investigation of the design space and to find optimum solutions in hybrid surrogate optimization settings [280–282], where we are able to exploit information coming from both the original model and its surrogate. Therefore, reliable surrogate models with selective predictions (i.e., they are provided only under high-confidence conditions) are essential to ensure the high quality of these accelerated aircraft design optimizations.

## 8.2 Approach

In this section, we describe the development steps required to codesign smoothness-based OOD detectors with FNN surrogates. In Figure 8.1, the proposed systematic workflow of the SmOOD approach is shown, including the OOD recognition, the training and testing of the

inherent regression and classification models.

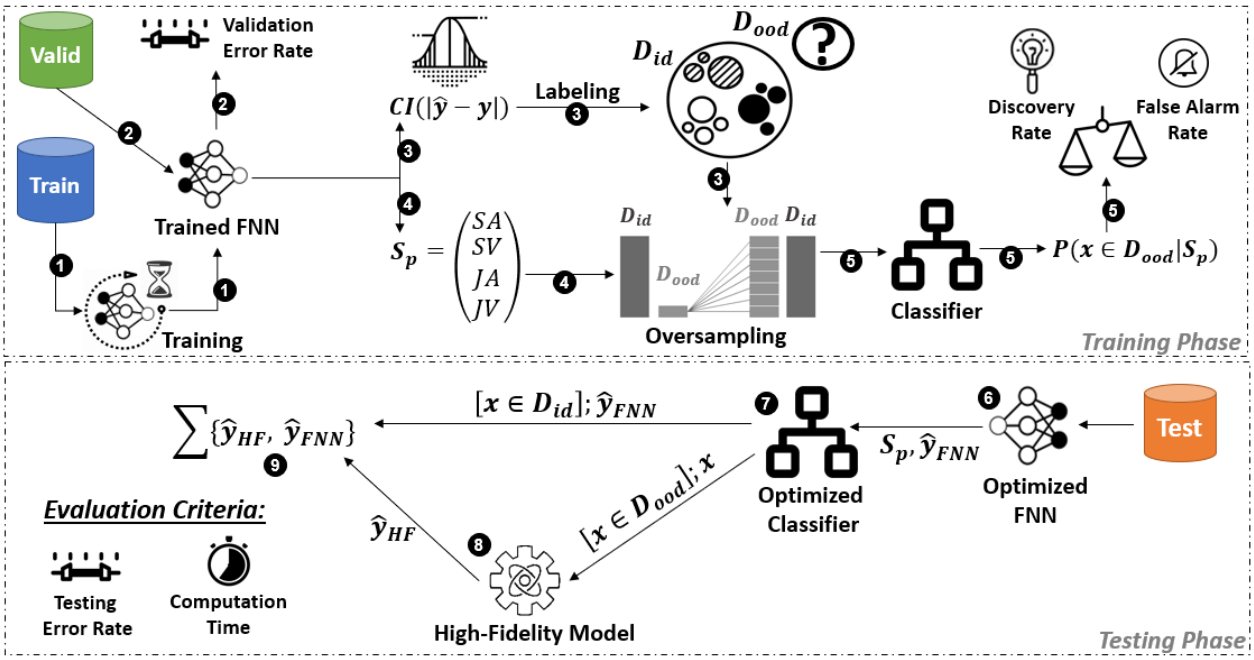



Figure 8.1 Schema of *SmOOD*: Co-Design of FNN Surrogate and Classification-based OOD Detection Models.

### 8.2.1 Characterization of Out-of-Distribution

In supervised learning, statistical models are commonly estimated via empirical risk minimization (ERM) [25], a principle that considers minimizing the average loss on observed samples of data, as an empirical estimate of the true risk, i.e., the expected true loss for the entire input distribution. Unfortunately, ERM assumes that training and test data are identically and independently distributed (a.k.a. i.i.d. assumption). Distributional shifts often occur in real scenarios for many reasons, such as domain transition, temporal evolution, or selection bias, which degrades the model's performance since certain captured correlations may not hold on these shifted inputs. Even worse, several research works [74, 87, 88] demonstrated that the optimized models can fail dramatically when involving strong distributional shifts. In many surrogate modeling cases where the HF simulations are expensive, selection bias is almost unavoidable, which ruins the i.i.d. assumption. The design of OOD detection methods for selective prediction is of more critical significance than reducing empirical risk further on the collected training samples. The first step consists of the OOD inputs characterization. In the context of aircraft surrogate performance models, we aim

to characterize the foreseeable design configurations that can be sampled during the design optimization, however, they may deviate from the training distribution. As explained above, ERM optimizes the model by minimizing the average loss, which leads to greedily absorbing the correlations and patterns that hold on the majority of training instances. Hence, ERM can produce biased models that are susceptible to outliers, unfair to minor subsets of data, or prone to out-of-distribution samples [86]. Thus, it is common to use held-out validation data as a proxy for unseen data points, aiming at more realistic estimates of the true risk. As shown in Figure 8.1 (step 3), the validation error rate may reveal the model's inefficiencies in terms of remaining high-error test inputs on which the learned patterns could not generalize, and they can be used for approximating the boundaries of the in-distribution. We rely on thresholding over the prediction confidence interval, i.e., the margin of validation errors at a certain level of confidence, and labeling the data point as OOD if its associated error exceeds the estimated margin.

### 8.2.2 Computation of Local Sensitivity Profiles

HF mathematical models are typically based on conservation laws and solve a coupled non-linear system of partial differential equations on a discretized spatio-temporal domain. This allows numerically stable simulations of the aircraft via their approximations in smooth or piecewise smooth dynamics. Hence, their NN-based surrogate counterpart must meet this apriori of smoothness in regards to the predicted quantities of interest in order to be a viable alternative to the HF model. In fact, the smoothness of the FNN's mapping function affects the model complexity as evidenced by the fact that smooth deep neural networks tend to generalize better than their less smooth counterparts [283]. Nonetheless, rigid smoothness techniques that constrain the learned function excessively by forcing it to be equally smooth throughout the input space, may throw away useful information about the input distribution [284]. As alternatives to these rigid methods, DL practitioners can employ regularization techniques, such as weight decay [285], dropout [286], and early stopping [155], to encourage smoothness of the model and improve generalization. Their hyperparameters should be tuned to achieve the desired level of smoothness that allows the network to allocate capacity as needed to maintain useful diversity, handle input modalities, and capture task-relevant information. In our study of OOD detection for aircraft surrogate performance FNNs, the smoothness is one of the fundamental properties of system design. Apart from regularizing the FNNs to smooth mapping functions, we created pointwise sensitivity profiling to capture the local function smoothness around the neighbor regions of each evaluated data point. On unseen data points, the integrity of the surrogate network can then be ensured by comparing their triggered pointwise sensitivity profiles to the degree of smoothness observed on the op-

timized DL model on the in-distribution inputs. In the following, we introduce the proposed pointwise sensitivity profiling for FNN that quantify different aspects of the network's output variations in relation with changes in input variables.

**The expected deviations of the network's output:**

The straightforward way to estimate the sensitivity of the neural network at an individual data point, is to assess the induced output deviations in response to random noise injected into the input features. According to Mi et al. [131], network sensitivity to input perturbations can be used as a surrogate for uncertainty. More precisely, they have proved that sensitivity and uncertainty have a nonnegative correlation in a setting of dense regression networks. Therefore, we include the following statistics on the output deviations under constrained input perturbations in the sensitivity profiling that will be used to separate between OOD and ID inputs.

Let $\Delta x$ be the perturbation applied to the input of $f$, a neural network, $\delta$ be the maximum threshold of the absolute value of $|\Delta x|$, and $\Delta y$ be the responding output deviation, $|f(x + \Delta x) - f(x)|$. Then, the two deviation-based sensitivity profiling metrics are defined respectively:

$$\mathrm{SA}(x) = \mathrm{AVG}(|\Delta y|) = \mathbb{E}_{\Delta x \sim \mathcal{U}(-\delta,\delta)}[|f(x + \Delta x) - f(x)|]$$

$$\mathrm{SV}(x) = \mathrm{VAR}(|\Delta y|) = \mathbb{E}_{\Delta x \sim \mathcal{U}(-\delta,\delta)}[(|\Delta y| - \mathrm{AVG}(|\Delta y|))^2]$$

**The input-output jacobian norm:**

As the feedforward neural networks are differentiable models, we also investigate its sensitivity at a point through the computation of input-output jacobian norm. Novak et al. [283] have presented extensive experimental evidence that the local geometry of the trained function as captured by the input-output Jacobian can be informative of the prediction confidence at the level of individual test points, and that it varies drastically depending on how close to the training data manifold the function is evaluated. Thus, the computed jacobian norm is likely to be higher at shifted inputs in comparison with the training inputs. Nevertheless, the feedforward networks are typically based on ReLU activations, which makes them not continuously differentiable. The derivative of the mapping function may therefore fluctuate sharply at small scales, and hence, considering the Jacobian of a specific data point will have less meaning than considering the Jacobians of a subset of nearby data points [287]. Thus, we apply random perturbations on the data point to yield samples from its neighborhood. Then, we compute statistics on the vanilla Jacobians estimated at these noisy samples to be

included in our network's sensitivity profiling, as follows.
Let $\mathbf{J}(\mathbf{x}) = \partial \mathbf{f}(\mathbf{x})/\partial \mathbf{x}^{\mathbf{T}}$ be the input-output jacobian of a neural network, $f$, at the input $x$, and $||.||_F$ is the Frobenius norm. Further, we assume that $\Delta x$ is the perturbation applied to the input of $f$, a neural network, $\delta$ is the maximum threshold of the absolute value of $|\Delta x|$, and $x' = x + \Delta x$.

$$\text{JA}(x) = ||\text{AVG}(\mathbf{J}(\mathbf{x} + \mathbf{\Delta x}))||_F = ||\mathbb{E}_{\Delta x \sim \mathcal{U}(-\delta,\delta)}[\mathbf{J}(x + \Delta x)||_F$$

$$\text{JV}(x) = ||\text{VAR}(\mathbf{J}(\mathbf{x}'))||_F = ||\mathbb{E}_{\Delta x \sim \mathcal{U}(-\delta,\delta)}[(x' - \text{AVG}(\mathbf{J}(x')))^2||_F$$

As demonstrated in the step 4 of our defined workflow (Figure 8.1), we define the pointwise sensitivity profiles, denoted $S_p$, as a real-valued vector that concatenate the above senstivity-related statistics within the neighborhood regions of individual data points, $x$, which can be formulated as follows, $S_p(x) = [\text{SA}(x) \parallel \text{SV}(x) \parallel \text{JA}(x) \parallel \text{JV}(x)]$.

### 8.2.3 Training of ID/OOD Classifier

After being estimated, the pointwise local sensitivity profiles are fed into a ML-based classifier, specifically, a binary classifier. The latter is trained on the sensitivity profiles of validation data points, along with their derived labels, i.e., ID or OOD, on the basis of the expected confidence interval for the validation errors. Nevertheless, the average and the margin of error are two metrics that will drive the development and the selection of surrogate models. Hence, it is reasonable to expect that these measurements are well optimized and only a small fraction of data points would remain with relatively high error over successive validation steps, which results in an imbalanced dataset of sensitivity profiles that is predominantly composed of ID examples with only a low percentage of OOD examples. To deal with class imbalance, we explore several commonly-used oversampling techniques [288] in the literature that produce synthetic data points belonging to the minority class to emulate a semblance of balance to the dataset. These techniques are necessary means of increasing the sensitivity of a classifier to the minority class, allowing the detection of as many OODs as possible from incoming inputs. This is shown in-between the step 4-5 of the SmOOD co-design workflow in Figure 8.1. By interpreting OODs as positive and IDs as negative, we should conduct an in-depth performance assessment of the ML-based classifier with aim of converging to optimal balance (i.e, the evaluation step 5 in Figure 8.1) between OOD discovery, referring to the ratio of the truly detected OODs among the actual OOD collection, and OOD false alarms, defined as the ratio of false positive arising from the actual ID samples. In the context of hybrid surrogate design optimization, a high OOD discovery rate ensures

trustworthy selective surrogate predictions where only ID samples are submitted to the surrogate model, and hence, the accelerated design optimization converges to high potential and feasible aircraft design configurations. On the other hand, a low OOD false positive rate is imperative to preserve the advantages of surrogate modeling and avoid useless, costly queries to the HF model.

### 8.2.4 *SmOOD*: Integration and Evaluation

During the training phase of the SmOOD workflow (Figure 8.1), the surrogate model is learned first, then its predictive errors are computed on the held-out validation datasets. Next, we leverage the confidence intervals of the estimated errors during the validation to identify the out-of-distribution (OOD) data (i.e., inputs on which the best-fitted model still produces relatively high errors). In parallel, we calculate the local sensitivity profiles for all the validation inputs. Since the best-fitted model was selected for its predictive performance, OOD inputs would have very small sensitivity profiles compared with their counterparts assigned to in-distribution (ID) inputs, and an oversampling procedure is required to amplify synthetically their occurrences in order to analyze the differences in smoothness-related characteristics between both groups of profiles. As a next step, we feed the oversampled sensitivity profiling data into a binary classifier that learns to distinguish between OOD and ID profiles. For optimal classifier selection, we assess both the OOD discovery and false alarm rates using k-fold cross-validation.

For the testing phase in the SmOOD workflow (Figure 8.1), test data assembles aircraft design configurations for which an assessment has been requested. Since SmOOD is a model-dependent OOD detection strategy, we first pass all the test inputs by the optimized FNN to determine their predictions and their corresponding local sensitivity profiles. Then, the optimized classifier serves as a calibrated OOD detector that generates a risk score for each precomputed sensitivity profile, quantifying the likelihood that its source test is indeed an out-of-distribution sample from the FNN surrogate's perspective. The next step depends on a criterion that evaluates whether the risk score falls below a predefined threshold (i.e., a default of 0.5 out of $[0, 1.0]$ score ranges), and two scenarios are then possible. If the criterion is satisfied, the surrogate prediction is returned and no further action is taken. If not, the test sample is sent to the High-fidelity model for simulation and its outcome is returned. Quality of results is assessed using two evaluation criteria. First criterion is the decrease rate achieved in respect to the prediction errors on testing samples, and especially, the further error reduction by the hybrid mode. A reliable OOD detection method must route all high risk samples to the HF model, reducing error rates significantly. Second, the computation time of hybrid mode should be much shorter than the pure HF-driven optimization since

this was the main purpose of setting up the surrogate model in the first place. Indeed, a well-calibrated OOD detection method should spawn a low rate of false alarms because these false positives would cause unnecessary HF simulations, slowing down the progress towards design optimization.

## 8.3 Evaluation

In this section, we first introduce the three aircraft performance case studies, as well as our evaluation setup, metrics, and methodology. Next, we evaluate *SmOOD* against standard baseline in terms of predictive performance and computation runtime.

### 8.3.1 Experimental Setup

This section details the different elements that were set up for SmOOD performance assessment, including aircraft design study cases, inherent models, baseline, as well as evaluation environment, strategy, and metrics.

#### Case Studies

Below, we briefly describe the studied aircraft performance factors [289], the influencer design variables [289], and the design of experiments used for dataset collection.

**Maximum TakeOff Weight (MTOW).** It represents the maximum weight at which the pilot of the aircraft is allowed to attempt to take off given its structural design. MTOW is an important factor in aircraft design. A higher MTOW means the aircraft can take off with more fuel and has a longer range, which makes it more appealing to customers. However, MTOW is subject to several structural constraints during the design optimization process.

**Time To Climb (TTC).** Climbing is the act of increasing the altitude of an aircraft. The main climb phase is the increase of altitude following the takeoff to reach the cruise level. As a way to measure an aircraft's climb performance, it is common to set up a reference cruise altitude level, then, estimate the time needed to climb to the predetermined altitude at a constant airspeed. This climb performance measurement is called the Time To Climb (TTC).

**Balanced Field Length (BFL).** It refers to the shortest runway length at which a balanced field takeoff can be performed by an aircraft design while complying with safety regulations. A balanced field takeoff occurs when the required accelerate-stop distance is equal to the required takeoff distance. Accelerate-stop distance is the runway length required by an aircraft to accelerate to a specific speed, and then, in the event an engine fails, to stop safely

on the runway. Thus, aircraft designers are senstitive to BFL as any changes in this require ensuring safety margins at takeoff are still respected.

**Design Configuration.** A set of 15 design variables are considered in the surrogate modeling based on experts' judgment on their influence on the above-mentioned quantities of interest. The design variables capture the wing design (i.e., aspect ratio, taper ratio, thickness ratio, and winglet span ratio), the fuselage geometrical structure, the engine thrust, and the mass of aircraft parts.

**Design of Experiment(DoE).** Our engineering collaborators configure the appropriate design of experiments (DOE) to collect the labeled data points. DoE assembles a set of tests that exercises the HF model across diverse design configurations to gather HF simulations. Indeed, the Latin Hypercube Sampling technique [290] is leveraged to efficiently sample from large, multivariable design spaces. LHS uniformly divides the range of each design variable into the same number of levels. It then systematically combines independent samples of the levels of each factor to specify a variety of random data points in the design space. Running the HF model on these data points gives us HF simulations that map design factors to their corresponding responses, i.e., the measurable quantities of interest. Therefore, we obtain 2259 of training data points and 1960 of testing data points for each one of the study cases.

### Models

**Surrogate.** Our base nonlinear regression model is a three-layer feedforward neural network(FNN) [291] that is trained using the Mean Squared Error (MSE) loss function with L2-norm regularization. Rectified linear units (ReLU) are used as hidden layer activation functions, and Adam is used as the optimization algorithm. In regards to the architecture, we followed the design principle of pyramidal neural structure [136], i.e., from low-dimensional to high-dimensional feature spaces/layers, as well as these dimensions are powers of 2 to achieve better performance on GPUs [264]. Regarding the hyperparameters tuning, we leverage the grid-search strategy and 5-fold cross validation to sample and try several possible settings. To outline their ranges, we denote $linspace(a, b, n)$ to indicate the range of $n$ equi-spaced values within $[a, b]$ and $logspace(c, d, base)$ to indicate the interval of $base^c, .., base^d$, where $c < d$. The FNN's width of layers are selected from $logspace(5, 10, 2)$, then, the optimizer's hyperparameters were tuned as follows: learning rate $\eta \in s \cup 3 \times s$, weight decay $\lambda \in s \cup 5 \times s$, where $s = logspace(-4, -1, 10)$. Batch size was tuned in $logspace(3, 7, 2)$, and epochs count in $linspace(50, 500, 50)$.

**OOD Detection.** Several ML classifiers (Logistic Regression [292], SVM [293], Random Forest [294], and Gradient Boosting [295]) and over-sampling techniques (SMOTE [296],

BorderlineSMOTE [297], ADASYN [298], and SVMSMOTE [299]) have been evaluated on our OOD detection problems using 5-fold cross validation process. We have found that Gradient Boosting and SVMSMOTE are the best design choices. Then, we follow the same hyperparameters tuning method used for FNN, i.e., grid-search strategy and 5-folds cross validation. For the gradient boosting classifier, we examine the learning rate, $\eta$ and the number of estimators,$n$, selected from the following ranges, respectively, $s \cup 5 \times s$ and $[100, 250, 500]$, where $s = logspace(-3, -1, 10)$ and $[100, 250, 500]$. In regards to SVMSMOTE oversampler, we consider $k \in [5, 10, 15]$, where $k$ represents the count of nearest neighbors to used to construct synthetic samples, and $r \in [0.25, 0.5, 0.75, 1.0]$, where $r$ indicates the desired ratio of the number of samples in the minority class over the number of samples in the majority class after resampling.

**Baseline**

**Surrogate.** We compare our FNNs with Gaussian Process(GP) regression that is already well researched for surrogate modeling [300], replacing expensive high-fidelity aircraft simulations. A GP [301] is a generalization of the Gaussian distribution to describe universal functions $f(x)$. The main ingredient of GP design is the selection of the covariance function $k(x, x')$, also called kernel. For our comparison with FNN, we are interested in approximating multiple quantities using the baseline. We chose the radial basis function (RBF) [302] that has been implemented to interpolate several aircraft design data [274, 282]. RBF [302] is an universal kernel function that can be used to fit any complex non-linear regression data. The hyperparameters of the kernel are optimized during fitting by maximizing the log-marginal-likelihood (LML) on the training data. As the LML may have multiple local optima, we set up 10 to be the number of repetitive restarts of the optimizer in order to improve the convergence towards optimal results.

**OOD Detection.** A basic approach of hybrid GP [282] for surrogate aircraft design optimization mimics a rule of thumb in engineering analysis: if the new data is reasonably similar to the existing one, it can be assumed to be reliable. Given a design configuration to assess, one can compute the standard deviation, $\sigma$, of the differences between the output predicted by RBF, $\hat{y}$, and the actual outputs, $y_1, y_2, ..., y_n$, of a set of $n$ neighboring configurations. The standard deviation $\sigma$ can be computed with differences $\|\hat{\mathbf{y}} - \mathbf{y}_i\|$ with $i \in [1, n]$, as follows,$\sigma = \text{std}\left([\|\hat{\mathbf{y}} - \mathbf{y}_1\|, \|\hat{\mathbf{y}} - \mathbf{y}_2\|, \|\hat{\mathbf{y}} - \mathbf{y}_n\| \ldots]\right)$. Then, the obtained standard deviation is compared to a threshold, $\sigma_t$. If it is lower, $\sigma < \sigma_t$, the prediction of RBF surrogate is adopted; otherwise, the HF model must be requested instead. Regarding the threshold value t, a priori statistical analysis can be performed on the validation data to derive the thresh-

old value that includes 95% of configurations. Using a 5-fold cross validation, we tune the number of neighbors considered in the computation of standard deviation within the range of $linspace(4, 20, 4)$, and we select the number that yields the highest correlation between the estimated standard deviation and the actual error.

### Evaluation Procedure

To evaluate SmOOD's effectiveness, we conducted quantitative and qualitative analyses. In the following, we detail the metrics and the procedure used.

**Quantitative.** Due to the high cost of running HF simulations, we adopt a 10-fold cross-validation method for all experiments in order to have different splits for the training and validation datasets and quantify the target metrics by averaging their values over the 10 iterations. Besides, all the included estimated metrics like error rates and runtime, are computed as average values over 5 runs or more, in order to mitigate the effects of randomness inherent in statisical learnign algorithms. To obtain domain experts' feedback, we interview, separately, two senior aircraft engineers from our industrial partner, Bombardier Aerospace. Next, we summarize their opinions and comments in written paragraphs. We then submit them for approval to ensure they are in alignment with what their claims are. Final paragraphs are added to each related research question. Below, we introduce the different evaluation metrics that have been used in the empirical evaluations.

**NRMSE.** stands for normalized root mean square error, and it is scale-independent version of RMSE that allows comparison between models at different scales. Indeed, RMSE is the average deviation between predictions and actual outputs, measured on the same scale and with the same output unit, and can be formulated as follows: $RMSE = \sqrt{\sum_{i=1}^{n} (y_i - \hat{y})^2 / n}$, where $y_i$ is the ith observation of y and $\hat{y}_i$ its corresponding prediction by the model.

In NRMSE, the expected model deviations are reported relative to the overall range of output values, and the formula becomes $NRMSE = RMSE/(y_{\max} - y_{\min})$.

**Confidence Interval (CI).** indicates the degree to which an estimate is expected to vary from the average, within a certain level of confidence. For example, a 95%CI of a model's error represents the upper and lower bounds within which the estimate of error will fall for 95 percent of the time (i.e., 95 samples out of 100 were taken). We use the bootstrapping method to produce accurate CIs because its non-parametric nature allows it to be used without making any prior assumptions about the estimate distribution.

**Precision.** represents the proportion of the positive identifications that were actually correct in a binary ML classification problem.

**Recall.** measures the proportion of actual positives that were correctly classified in a binary ML classification problem.

**Macro-Averages of Precision/Recall.** refer to the arithmetic averages of the precision and recall scores of individual classes.

**Precision-Recall (PR) curve.** is a useful evaluation plot for binary classifiers that return class membership probabilities, when there is moderate to large class imbalance [303]. It simply plots the precision versus recall obtained by a classifier using different probability thresholds.

**Area Under Precision-Recall (AUPR).** approximates the area under the PR curve, ranging from 0.0 to 1.0.

**%Decr_Err.** The decrease ratio of error we achieve by a certain improvement, can be formulated as follows:

$$\%Improv\_Err = \frac{\textit{pre-NRMSE} - \textit{post-Err}}{\textit{pre-NRMSE}} \times 100 \tag{8.1}$$

Where *pre-NRMSE* and *post-NRMSE* refer to the actual NRMSE and the reduced NRMSE.

**Speed up.** is a popular measure for the relative performance of two systems processing the same problem. In our case, we denote $T_s$ be the surrogate compute time, and $T_o$ the High-fidelity compute time. Then, the speedup due to surrogate modeling can be computed as follows, $S_{pure} = T_o/T_s$. Given our OOD detection strategy used in hybrid surrogate optimization, we obtain $p$, as a proportion of instances that can be predicted by the surrogate. Then, the remaining $1-p$ proportion of instances that require requests to HF model. This means that the speed up of the hybrid surrogate model can be formulated as follows, $S_{hybrid} = T_o/(T_d + p_s + (1-p) * T_o)$, where $T_d$ is the total computation time needed to predict whether each of the inputs is OOD or not.

**Qualitative.** In order to obtain the input of domain experts, we interviewed two senior aircraft engineers with more than 10 years of experience, who work with our industrial partner, Bombardier Aerospace. Both of them are proficient in the use of MDO in aircraft design. Furthermore, they are part of the team that developed the high-fidelity physics model; therefore, they are familiar with the QoIs associated with the case studies.

As a first step, we present the quantitative analysis to both aircraft engineers separately so that they can get detailed performance metrics, as well as all execution costs for all model development steps (training, tuning of hyperparameters, etc). Then, we ask them to provide a critique of the significance of the obtained performance metrics from an aircraft design standpoint. Specifically, we seek comparisons of FNN surrogates along with our co-designed OOD detection method, as opposed to conventional surrogate modeling setups with GP and uncertainty quantification. In addition, we request explanations on how SmOOD contributes to quality assurance and acceleration of data-driven aircraft design optimization. Afterwards, we compile and merge their opinions and comments into paragraphs. Next, we submit them

for approval to ensure their claims are reflected in the written content, and that they both agree with the conclusions made. Final consensus statements are added as feedback from domain experts to each related research question.

**Environment**

We use Pytorch [265], an established DL framework for modeling and training feedforward neural networks. We leverage GPy [304], a popular and maintained framework to design and train Gaussian processes. In terms of the hardware environment, we use CPU machines for all the experiments for fairness comparisons. The HF models were running on a machine with Intel Xeon CPU E5-1630 v3 of 3.5Ghz and a 32 Gb of RAM. The FNNs and GPs were running on Standard Virtual Machines using 4 cores on Intel Xeon Platinum 8168 CPU of 2.70GHz and 8 Gb of RAM.

### 8.3.2 Experimental Resuls

In conducting the evaluation of the proposed approach, we studied the following research questions:

**RQ1. Is FNN a viable general-purpose approximator to model complex surrogate aircraft design performance models?**

**Motivation.** The objective is to evaluate the effectiveness of FNN as a universal approximator for surrogate aircraft design performance models in comparison to GP, which is the mainstream universal estimator for surrogate modeling. **Method.** We train and tune both

Table 8.1 Performance Metrics for different pairs of QoI and Surrogate Model.

| QoI | Model | Valid_Err | Test_Err | Valid_CE@99%CL | Test_CE@99%CL | %Valid_OODs | %Test_OODs |
|---|---|---|---|---|---|---|---|
| **MTOW** | *GP* | 0.0415 | 0.0363 | [0.0212,0.0596] | [0.014, 0.0547] | 5.14% | 4.39% |
| | *FNN* | 0.0353 | 0.0319 | [0.0163, 0.0523] | [0.0111, 0.0486] | 1.59% | 1.17% |
| **TTC** | *GP* | 0.1707 | 0.1698 | [0.1646, 0.1769] | [0.1632, 0.1762] | 40.95% | 41.33% |
| | *FNN* | 0.0764 | 0.0803 | [0.0662, 0.0865] | [0.0668, 0.0942] | 5.0% | 5.26% |
| **BFL** | *GP* | 0.0454 | 0.0439 | [0.0241, 0.0637] | [0.0168, 0.0664] | 6.64% | 5.97% |
| | *FNN* | 0.0435 | 0.0361 | [0.0198, 0.0637] | [0.0124, 0.0599] | 1.42% | 1.12% |

FNN and GP surrogate models on all the three aircraft design performance factors. Then, we test the optimized models on a held-outs test dataset to assess their predictive performance. As performance metrics, we compute the NRMSE on validation datasets to gauge the fitness quality of both models and the NRMSE on test dataset to compare their generalizability

capabilities. Besides, we infer the CI of each estimate of error with 99% of confidence level, as an accurate risk assessment of the possible range of the model's prediction errors, and especially, the upper bound that indicates how severe the error is expected to be. Last, we calculate the ratio of OOD inputs observed on validation and testing datasets according to our OOD labeling rule (Section 8.2.1), i.e., their actual prediction errors are higher than the maximum expected margin.

**Results.** Table 8.1 summarizes the obtained performance metrics for each pair of regression problem and surrogate model type. The results show that FNNs have reached lower validation and test estimation errors with even tighter confidence intervals than GP in all the studied surrogate modeling problems. Moreover, the ratio of model-dependent OOD inputs that have been experienced by GP is substantially higher than FNN. This demonstrates that GP's high average error and wide confidence interval are due to the prevalence of complex design configuration inputs on which GP fails and cannot provide a reliable assessment. It is on the time-to-climb (TTC) design problem that the gap between the two types of surrogate models is most pronounced. There are, indeed, high nonlinearities in TTC modeling, where two numerically-close inputs can result in distant takeoff behaviors. Because GP kernels are biased towards their predefined distributional priors and have a limited learning capacity in comparison to FNN, they do not generalize well to aircraft performance design problems with dispersed behaviors, such as the time to climb simulations, ranging from takeoff failures to accelerated climbing scenarios.

**Finding 1:** Feedforward neural network is a viable universal approximator, outperforming Gaussian Process, in building complex, highly-nonlinear surrogate aircraft performance models.

**Domain Expert Feedback.** Aircraft engineers find the comparison results enlightening as they demonstrate that data-driven surrogate modeling is still a viable solution even in cases of high nonlinearity input-output mappings, such as TTC models. Moreover, GP's results on TTC are expected, since it is the origin of this initiative on reliable FNN surrogates. It was previously necessary for aircraft engineers to re-run the design optimization with the HF model in the objective function because the GP gives inaccurate TTC predictions and hinders the convergence of the optimizer, which does not find a feasible design.

**RQ2. Can local sensitivity profiles capture relevant information on FNN behaviors to accurately signal OODs across generated samples?**

**Motivation.** The aim is to determine how well pointwise local sensitivity profiles can discriminate between ID and OOD inputs compared with exploiting the uncertainty estimation of GP.

**Method.** For GP, the predictions, $\hat{y}$ are made in a probabilistic way, i.e., described by mean, $\mu_{\hat{y}}$, and standard deviation, $\sigma_{\hat{y}}$, which is supposed to reflect the epistemic uncertainty, i.e., how certain the model is with respect to its prediction. We therefore statistically compare the distribution of $\sigma_{\hat{y}}$ on ID versus OOD examples discovered during the validation. To do that, we compare their box plots and perform Mann-Whitney U nonparametric hypothesis tests to determine whether the $\sigma_{\hat{y}}(\mathcal{D}_{id})$ and $\sigma_{\hat{y}}(\mathcal{D}_{ood})$ were sampled from the same population. Regarding FNN, we compute local sensitivity profiles triggered by FNN on the validation data points, and then analyze the distribution of these profiles among the ID and OOD groups. To do that, we use Principal Components Analysis (PCA) to visualize the computed multivariate local sensitivity profiles into 2D graphs. Indeed, we reduce the dimensions into two orthogonal principal components that preserve the maximum amount of variance explained by the original multivariate points, as well as, we add the original variable vectors to show their correlations and directions w.r.t the principal components. Moreover, we colored the ID and OOD points with different colors to identify the differences between the two groups. In addition, we perform Mann-Whitney U tests for each of the sensitivity profile variables to statistically confirm if the ID and OOD groups are likely to yield values originating from two different distributions.

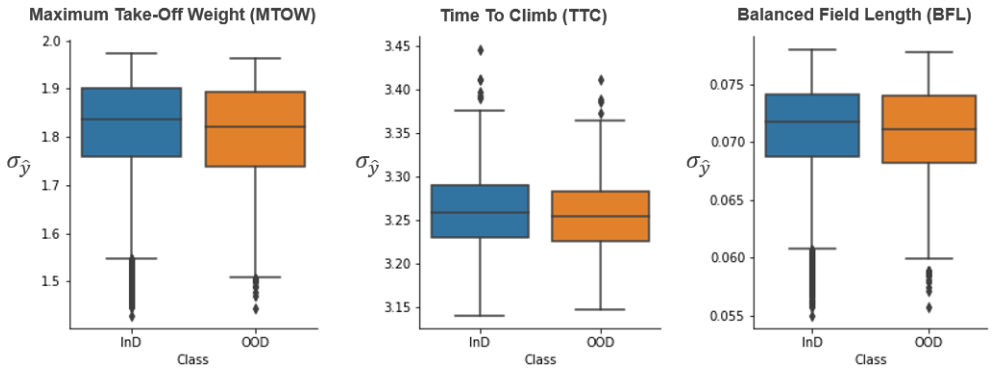



Figure 8.2 Comparison of GP's standard deviation distribution between ID and OOD samples for each QoI

**Results.** Figure 8.2 shows the box plots of the GPs' standard deviations for both ID and OOD examples. As can be seen, the groups' boxes have very close sizes and almost overlap completely for all the regression problems. Additionally, hypothesis testing suggests no statistically significant differences between the uncertainty estimates predicted by GP on ID

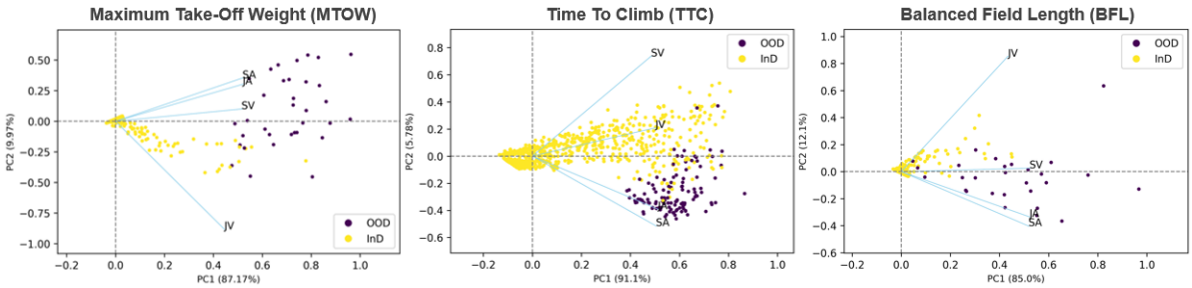



Figure 8.3 Graph of Pointwise Sensitivity Profiles for $\mathcal{D}_{id}$ and $\mathcal{D}_{ood}$ w.r.t the derived 2 Principal Components

versus OOD. The median lines in the box plots indicate that the median of the standard out-of-distribution sample is lower than the standard identification sample, demonstrating how GP incorrectly produces a sense of overconfidence about out-of-distribution samples. Even worse, the comparison of the median lines within the boxes highlights that the median of $\sigma_{\hat{y}}(\mathcal{D}_{ood})$ is lower than the median of $\sigma_{\hat{y}}(\mathcal{D}_{id})$, which shows how GP incorrectly tends to produce overconfident epistemic uncertainty when confronted with out-of-distribution samples. This is definitely an expected degradation in the reliability of uncertainty estimates, since the posterior distributions of predictions were calibrated using available samples, mostly ID, suggesting bias against distant examples. In contrast, Figure 8.3 present 2D plots of PCA analyses based on the sensitivity profiles of the regression models for the studied QoIs, MTOW, TTC, and BFL. As can be observed, the ID examples are grouped together and form a sort of cluster, whereas the OOD examples are quite dispersed and far from the centroid of the ID group. The hypothesis testing results confirm these observations, and there are statistically significant differences (with $p\text{-}value < 1e-6$) between the ID and OOD data groups in regards to all of the sensitivity profile's variables. Furthermore, the high dispersion and rarity of the OOD data points highlight that the identified scenarios represent corner-case and extreme behaviors with diffent patterns, rather than representing a novel distribution.

**Finding 2:** Pointwise local sensitivity profiles can accurately separate OOD and ID samples using FNN, whereas the reliability of GP's epistemic uncertainty declines on OOD samples.

**Domain Expert Feedback.** Aircraft engineering experts confirm they have experienced overconfidence of GP uncertainties before, especially when using objective functions in MDO that include the variance to better steer the search to high potential regions. In that case, over-confident variance is misleading to the optimizer. They perceive the added value of

the FNN sensitivity profiling, which is able to capitalize on the smoothness of input-output mapping as it is an a priori system property and an implicit assumption in the HF model's differential equations.

**RQ3. How effective is *SmOOD* in the coverage of OOD inputs across generated samples?**

**Motivation.** The goal is to assess the performance of *SmOOD* when co-designed with FNN against the baseline strategy used with GP in terms of precision and coverage.

**Method.** We train and tune our gradient boosting classifier on the local sensitivity profiles yielded by FNNs during the validation for each regression problem. Then, we test the optimized classifier on the held-out testing dataset. Next, we draw PR curves along with their associated AUPR scores for each studied problem. Concerning the OOD detection baseline for GP introduced in Section 8.3.1, it provides only labels with no class scores, we compute the precision and recall scores for each class, separately, and their macro average for each pair of surrogate model and OOD detector.

Table 8.2 Performance Comparison of SmOOD and Base for OOD Detection.

| Target | Detector | Model | Class | Precision | Recall |
|---|---|---|---|---|---|
| **MTOW** | *Base* | *GP* | ID | 96% | 96% |
| | | | OOD | **11%** | **10%** |
| | | | MA | 53% | 53% |
| | *SmOOD* | *FNN* | ID | 100% | 100% |
| | | | OOD | **72%** | **91%** |
| | | | MA | 86% | 95% |
| **TTC** | *Base* | *GP* | ID | 57% | 91% |
| | | | OOD | **14%** | **2%** |
| | | | MA | 36% | 47% |
| | *SmOOD* | *FNN* | ID | 100% | 98% |
| | | | OOD | **70%** | **93%** |
| | | | MA | 85% | 95% |
| **BFL** | *Base* | *GP* | ID | 94% | 96% |
| | | | OOD | **13%** | **10%** |
| | | | MA | 54% | 53% |
| | *SmOOD* | *FNN* | ID | 100% | 100% |
| | | | OOD | **76%** | **73%** |
| | | | MA | 88% | 86% |

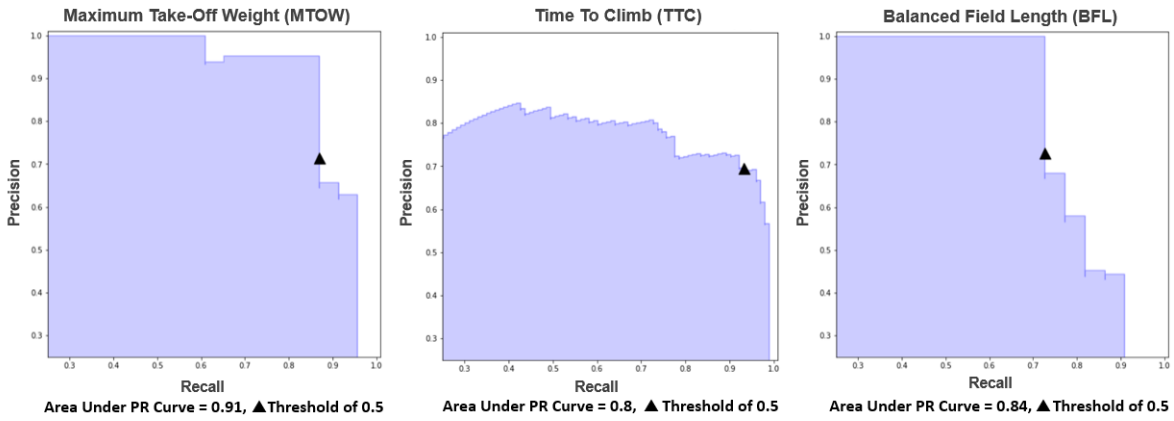



Figure 8.4 Precision-Recall Curves obtained by the optimized binary classifier for each QoI

**Results.** Figure 8.4 shows the precision-recall curves obtained by FNN for each aircraft performance problem and their associated AUPR scores. As can be seen, *SmOOD* was effective in fitting the collected local sensitivity profiles as evidenced by the relatively high AUPR scores. Although some levels of precision could be sacrificed to increase recall, we believe it is better to leave the default threshold at 0.5 since it yields a good trade-off between precision and recall, as shown by the marked points on the PR curves. Further pre-tuning of thresholds on validation dataset can lead to overfit the validation dataset and inversely causes a degradation of the classifier's performance. Besides, Table 8.4 summarizes all the classification scores: both of precision and recall for each class, as well as their macro-averages for each pair of surrogate model type and aircraft design performance problem. Our method outperforms by far the baseline approach. This demonstrates the lack of generalizability expressed by the rigid smoothness apriori of the baseline OOD detection method, which could not hold for complex, highly-nonlinear aircraft design performance models. Our method, in contrast, retrieves pointwise local sensitivity profiles and fits a binary ML classifier to the high-fidelity simulations to serve as a calibrator and detects erroneous smoothness behavior triggered by the FNN during the inference on unseen test inputs.

**Finding 3:** *SmOOD* was effectively able to reveal at least 73% and up to 93% of the out-of-distribution examples on the different aircraft design performance models.

**Domain Expert Feedback.** Aircraft engineers emphasize the importance of developing such mitigation strategies against behavioral drifts of these black-box ML models to prevent their use under suspicious conditions. They point out that design optimization studies are actually driven by an end-to-end automated process, which results in one best configuration communicated to managers and related engineering teams. Thus, such 3-5% of unreliable

predictions can enable the accelerated MDOs to reach feasible solutions at an early stage, however, the inefficiency of these solutions might remain hidden for quite a while.

**RQ4. What are the benefits of deploying *SmOOD* in hybrid surrogate optimization settings?**

**Motivation.** We aim to examine the effects of *SmOOD* on the prediction errors and computation time when it is combined with the FNN for hybrid surrogate aircraft design optimization.

**Method.** We compute the NRMSE and inference runtime values for all the surrogate variants: GP, FNN, HybridGP (GP + Baseline) and HybridFNN (FNN + SmOOD). Then, we derive their SpeedUp relative to the HF model, as well as, the Decrease rate of NRMSE achieved by the hybrid versions w.r.t. the pure surrogate counterparts. As a way to compare the surrogate model types, we calculated, separately, the computation durations required for all the design and evaluation steps (training, tuning, validation, and testing) for each type of model. Furthermore, we compute the same computation times for the design of OOD detection strategies to determine the overhead added when using an OOD detector to switch between HF model and its surrogate counterpart in hybrid surrogate design optimization settings.

**Results.** According to Table 8.3, the inclusion of *SmOOD* with FNN surrogates contributed to significant decrease ratios in prediction error, from 12% to 47%, while the baseline leads to a maximum of 1.25% decrease rate on the GP's prediction errors. Hence, *SmOOD* was able to cover many problematic OOD inputs with high errors that lead to considerable reduction in FNN prediction errors. Besides, the speed up rates obtained by *SmOOD* plus FNN surrogates are mostly higher than those obtained by Baseline plus GP counterparts. Indeed, *SmOOD* was able to detect the underlying OODs more accurately with fewer false alarms than the baseline, which turns almost three hours of runtime to a few minutes. FNN surrogates have already better speed up in the inference time than their GP counterparts (see Table 8.3). Table 8.4 also reports lower values of training time, validation time and test time obtained by FNN surrogates compared to those yielded by GP counterparts. This can be explained by the FNN's deterministic mapping function that includes consecutive weighted sums and ReLU activations, and by its learning algorithm that applies loss gradients w.r.t. the parameters to iteratively update them. On the other hand, GP's mapping function is stochastic including multivariate posterior gaussian distributions, and its approximate Bayesian learning that updates our prior belief in our gaussian parameters in line with marginal log-likelihood esti-

Table 8.3 Improvement Evaluations for different pairs of QoI and Surrogate Model.

| QoI | Model | Test_ERR | %ERR_Decr | Runtime | SpeedUp |
|---|---|---|---|---|---|
| **MTOW** | HF | - | - | 02h:57m:58s | 1 |
| | GP | 0.0363 | - | 299.26ms | $3.57 \times 10^4$ |
| | HybridGP | 0.036 | **0.83%** | 07m:38s | $2.33 \times 10$ |
| | FNN | 0.0319 | - | 0.81ms | $1.32 \times 10^7$ |
| | HybridFNN | 0.0169 | **47.02%** | 02m:38s | $6.76 \times 10$ |
| **TTC** | HF | - | - | 02h:57m:58s | 1 |
| | GP | 0.1698 | - | 289.47ms | $3.69 \times 10^4$ |
| | HybridGP | 0.1677 | **1.24%** | 10m:48s | $1.65 \times 10$ |
| | FNN | 0.0803 | - | 15.51ms | $6.88 \times 10^5$ |
| | HybridFNN | 0.0441 | **45.08%** | 12m:31s | $1.42 \times 10$ |
| **BFL** | HF | - | - | 02h:57m:58s | 1 |
| | GP | 0.0439 | - | 290.07ms | $3.68 \times 10^4$ |
| | HybridGP | 0.0435 | **0.91%** | 08m:32s | $2.08 \times 10$ |
| | FNN | 0.0361 | - | 4.58ms | $2.33 \times 10^6$ |
| | HybridFNN | 0.0317 | **12.19%** | 01m:54s | $1.42 \times 10$ |

Table 8.4 Computation Times for different steps of Surrogate Model Design.

| QoI | Model | Train_Time | Tune_Time | Eval_Time | Test_Time |
|---|---|---|---|---|---|
| **MTOW** | GP | 02m:50s | - | 27m:28s | 299.26ms |
| | FNN | 00m:17s | 05h:41m:39s | 02m:53s | 0.81ms |
| **TTC** | GP | 04m:25s | - | 42m:07s | 289.47ms |
| | FNN | 02m:10s | 22h:49m:54s | 21m:04s | 15.51ms |
| **BFL** | GP | 03m:11s | - | 31m:12s | 290.07ms |
| | FNN | 00m:54s | 20h:43m:09s | 08m:46s | 4.58ms |

mates on the observed data. Nonetheless, Table 8.4 shows that only FNN surrogates require a time-consuming and hand-crafted hyperparameters tuning to select the structure of the model (i.e., depth and width), configure the optimizer, adapt the regularizer strength, etc. All these choices are guided by trial-and-error processes on validation set. In contrast, the GP surrogate design is straightforward, and the kernel hyperparameters are determined systematically within the Bayesian posteriori optimization. The tuning of FNNs is common for any DL solution; so GPU-enabled parallelism and multi-machine distribution is straightforward when using modern DL frameworks.

Table 8.5 demonstrates that *SmOOD* outperforms the baseline in terms of low design and evaluation workloads. The reason is the fast computation of predictions and derivatives

Table 8.5 Computation Times for different steps of OOD Detection Method Design.

| Method | Train_Time | Tune_Time | Valid_Time | Test_Time |
|---|---|---|---|---|
| **Baseline** | 0.32s | 178.99s | 3.17s | 271.08ms |
| **SmOOD** | 0.43s | 12.47s | 4.22s | 2.16ms |

using FNN surrogates to construct the local sensitivity profiles, and the ease with which the *SmOOD* inherent classifier and oversampler can be trained and tuned due to the low dimensionality of the precomputing sensitivity profiles. This is contrary to the baseline which requires repeated distance calculations between a requested design configuration and all the validation set to identify the nearest neighbors.

**Finding 4:** *SmOOD* plus FNN surrogate enables accelerated and accurate hybrid optimization, achieving, on average, 34.65% and 58.36× of decrease error rate and computation speed up rate.

**Domain Expert Feedback.** According to domain experts, as long as the tuning process is automated, the FNN is still viable given the achieved error rate regardless of the build time. Furthermore, they emphasize that these FNN surrogates are mapping aircraft design variables to a particular aircraft performance attribute. Hence, an optimized FNN surrogate can be involved in many design optimization studies, which compensates for its relatively-heavy creation procedure. Most changes applied to the HF simulations are adjustments to the design variables in order to accommodate new requirements recommended by marketing analysts. This may result in further training on FNN parameters rather than structural/hyperparameters tuning. Creating FNN surrogates from scratch can be only done in response to less-frequent, major updates to HF model inner functions and capabilities. Nevertheless, experts suggest taking full advantage of the hybrid mode and storing the detected OOD inputs to further fine-tune the FNN after every optimization cycle since the training is relatively fast.

## 8.4 Chapter Summary

In this chapter, we demonstrate that feedforward neural networks outperforms Gaussian processes across several surrogate models of aircraft design performance, especially when the predicted quantity is highly nonlinear. These high-capacity black box models are prone to out-of-distribution issues, which restricts their direct use in aircraft design optimization. Due to the risk of overconfident uncertainties against these OOD samples, SmOOD effectively

spots them by identifying their suspicious local sensitivity behaviors that are far from the level of FNN smoothness observed across validation. Statistically, SmOOD reveals up to 93% of OOD samples, and its use as router in hybrid surrogate aircraft design optimization leads to 34.65% and 58.36× of error reduction and runtime speed up rates. Our empirical evaluation reinforces our prior belief in the a priori smoothness that exists over the HF simulations data. Indeed, we expect similar priors to exist in many other applications, on which our findings can be extrapolated, and our local sensitivity profiling can serve as a surrogate for model uncertainty, and as a discriminatory criterion to separate ID from OOD.

# CHAPTER 9 CONCLUSION

In this chapter, we summarize our findings and conclude the thesis. In addition, we discuss the limitations of our proposed approaches and outline some directions for future work.

## 9.1 Summary

It is becoming more and more common to find deep-learning applications across different domains, including safety-critical ones. This growing attention is due to the impressive performance of state-of-the-art DL models on handling high-dimensional data and performing various complex tasks. However, their initial deployments in critical domains such as aircraft system engineering shed light on the many challenges of these disruptive self-learning capabilities to the software system reliability. Hence, their quality assurance is becoming a preeminent priority in both SE and AI communities to gain the confidence of domain experts in the statistical learning and build trustworthy DL applications. Below, we present a summary of the employed methodologies and elaborated solutions that we have discussed throughout this thesis.

**NeuraLint: A Static Rule-based DL Program Debugger.** Based on empirical studies on DL bugs, we extract the crash-inducing bugs that represent coding faults leading to raising an exception and stopping the learning. Then, we filter them, where the acceptance criterion is their footprints on code which can be statically detected by code synthesis and checking rules. Additionally, we identify poor design choices belonging to non-crashing DL bugs, but which satisfy the above criterion. Therefore, we build a meta-model for a typical DL program that includes its base skeleton and fundamental components invariably at the dependencies of coding libraries. This level-up abstraction paves the way for the application of model-driven checking rules for the detection of coding errors and inappropriate model structures in DL training programs. We build our debugging tool from scratch using static Python code analysis, and support most of the API routines provided by DL libraries: Tensorflow and Keras. The effectiveness of our debugging method is determined by its fault detection capability on synthetic and real-world buggy DL programs. Synthetic buggy DL programs represent the results of a base clean DL program plus an injection of a fault. However, the real-world buggy DL programs are extracted for SO posts and bug fixing commits of Github projects.

**TheDeepChecker: A Dynamic Property-based DL Program Debugger.** To complement our proposed static DL code debugger, we continue by extracting the silent bugs that represent coding faults and misconceptions leading to unstable learning dynamics and inefficient trained models. Then, we elaborate the common pitfalls in designing and implementing DNNs, while pointing out their associated non-crashing bugs. This is important to explain the connection between each considered coding bug or misconfiguration and its induced negative effect on the neural network's components. Next, we derive our property-based verification routines to capture dynamically violations of fundamental design principles and fine-grained training efficiency traits with the objective of revealing hidden bugs early on and away from repetitive costly training trials. The effectiveness of our debugging method is measured in terms of training pitfalls and issues detection capabilities for both types of DL buggy software, synthetic and real-world programs. We also assess the dynamic checking workload, and the usability of our approach and the ease of DL bug fixing with it through live sessions with DL engineers.

**DeepEvolution: A Search-based DL Testing Approach.** As a generic-purpose DL model testing method, we propose DeepEvolution, a search-based approach for metamorphic transformation generation to test modern DNNs across domains. Considering the application-specific requirements, we define a development process to derive semantically-preserving metamorphic relations that are capable of exposing the DNNs' failures against simulations of realistic conditions that can occur in practice regarding two possible types of change factors: (i) system-level factors that refer to deprecation of hardware such as camera lens distortions, microphone damages, and unknown words; (ii) context-level factors that represent the transformations arising naturally within the external environment in which the DNN is deployed such as changing lighting conditions, outdoor or outdoor environmental noises, and typing errors. Then, we codify these metamorphic transformations into a data-independent constrained space of their vectorized parameters and settings, which can be explored effectively by nature-inspired, population-based metaheuristics algorithms for a maximum disclosure of unexpected behaviors. In regards to test objectives, we define two behavioral drift fitness functions that capture fine-grained divergences in the scores yielded by a single DNN or equivalent DNNs of varying arithmetic precision. Hence, our searching metaheuristic algorithm finds the most fault-detecting test inputs that correspond: (i) unstable behaviors of optimized DNNs on synthetic inputs that expose inappropriate inductive biases;(ii) divergent behaviors between quantized DNNs and their original counterparts that reveal low-precision parameters inefficiency to preserve the relevant learned patterns. A rich set of case studies using architecturally-different recognition models trained on popular im-

age, speech, and natural language datasets are performed to evaluate the performance of our approach and demonstrate its generality.

**PhysicAL: Physics-based Adversarial Machine Learning.** Our proposed search-based DL testing method shows the potential of searching over designed metamorphic transformations to derive synthetic test inputs with high fault-revealing ability. The evaluation was done on classification neural networks because we can implement semantically-preserving transformations on their inputs that keep the associated label invariant. However, applying DeepEvolution to regression problems requires knowing the expected output changes respective to the designed input transformation. Therefore, we rely on the high-level physics specifications including first principles and *apriori* system design properties that can be expressed in the format of input-output sensitivity rules, in order to anticipate the directional expected trend of output under the input variations outlined in the premises of the rule. Our proposed testing method also applies search-based algorithms to explore effectively the neighborhood local input space around each original data aiming at revealing hidden physics inconsistencies of the DNN. Next, we design a physics-informed adversarial training technique that leverages the revealed to guide the parameters optimization routines towards learning better inductive biases with lowest amount of deviation errors and physics inconsistencies in its outputs. The effectiveness evaluation of our proposed testing approach was conducted on two industrial cases of aircraft system performance models for which the flight test data are costly to collect, but the physics grounded input-sensitivity can be derived by aircraft engineers.

**SmOOD: Smoothness-based Out-Of-Distribution Detector.** Even domain-aware DL testing still cannot guarantee 100% reliability of a DNN, especially when the predicted quantity is highly nonlinear and out-of-distribution data (OOD) can occur in production. Therefore, we decide to work on bounding the regions of in-distribution (ID) data points on which the optimized DNN is ensured to provide stable and trustworthy predictions. The trust boundary delimitation allows selective operation when the DNN is used as an assistance system or to replace more complex models such as surrogates. Due to the risk of overconfident uncertainties against these OOD samples, we also rely on foreknown apriori properties of the system to systematically separate ID and OOD samples. Specifically, we define pointwise sensitivity profiles that estimate the local smoothness of the DNN at a given data point, then, suspicious ones are identified by their unjustified divergent sensitivity behaviors w.r.t the observed levels of DNN smoothness across the validation phase. Our detection strategy is deployed as a surrogate/HF model switcher in hybrid optimization settings, where its main

goal is to request the HF model only when a given input is labeled as OOD for the surrogate model. This enables to decrease the surrogate prediction errors with a minimal overhead cost of HF calls. The assessment of our proposed local sensitivity profiles and smoothness-based OOD detection method, was conducted on three aircraft design variables study cases, along with comparisons to common baselines, respectively, gaussian uncertainty estimation, and relative deviations from top-k neighbors.

## 9.2 Limitations and Future work

- Our proposed static and dynamic debugging techniques are extensible by the design, however, the current versions support mainly the feedforward neural network architectures including dense networks and CNNs. In the future versions, we plan to enlarge the scope of DNN architectures and show the applicability of our techniques on most of the mainstream deep learning systems. Another direction of research for the future is to extend our debuggers by automatic repair mechanisms [305] that include code recipes (i.e., generic snippet of code, one or multiple lines of code, characterizing a particular fix of a coding bug, an ineffective implementation or a misuse of APIs) for common errors and inefficiencies in relation to, respectively, DL source code or training anomalies. The goal is to develop an automatic generation of fix suggestions for these prevalent bugs to the DL engineer.

- Despite our physics-based adversarial testing improving the iid evaluations by exposing the maximum amount of network inconsistencies, it represents a separate phase prior to the physics-guided adversarial training. Hence, we can think about another type of adversarial search tightly connected to the model regularization (i.e., finds the most effective AXs to improve the physics consistency of the model.); so the users can tune our approach on their models to the best consistency-error trade-off for the underlying engineering case studies. Moreover, our physics-guided adversarial ML approach can have profound implications for statistical model engineering. Therefore, we propose an autoML framework [306] that includes our physics-informed loss to steer the design and tuning of the DNN towards more suitable settings for solving the consistency-error trade-off.

- Although SmOOD is very efficient as a switcher in hybrid optimization settings, it stops at detecting potential OODs and requests the HF model instead of the surrogate FNN to make their assessments. However, these revealed OODs can be leveraged to patch the current version of FNN through fine-tuning its parameters on them aiming

at reducing further the HF requests in the future design assessments with the cost of a relatively low active learning [307] overhead.

# APPENDIX A CO-AUTHORSHIP

Earlier studies in the thesis were published/submitted as follows:

- On Testing Machine Learning Programs, Houssem Ben Braiek and Foutse Khomh, in *Journal of Systems and Software (JSS), 2020, Elsevier.*
- Automatic Fault Detection for Deep Learning Programs Using Graph Transformations, Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, Foutse Khomh, in *Transactions on Software Engineering and Methodology (TOSEM), 2021, ACM.*
- Testing Feedforward Neural Networks Training Programs, Houssem Ben Braiek and Foutse Khomh, in *Transactions on Software Engineering and Methodology (TOSEM), 2022, ACM.*
- Physics-Guided Adversarial Machine Learning for Aircraft Systems Simulation, Houssem Ben braiek, Thomas Reid, and Foutse Khomh, in *Transactions on Reliability, 2022, IEEE*
- DeepEvolution: A Search-based Data Transformation for Deep Learning Model Testing, Houssem Ben braiek, Ahmed Haj Yahmed, Rached Bouchoucha, Foutse Khomh, and Sonia Bouzidi, under review in *Transactions on Software Engineering and Methodology (TOSEM).*
- SmOOD: Smoothness-based Out-of-Distribution Detection Approach for Surrogate Neural Networks in Aircraft Design, Houssem Ben Braiek, Ali Tfaily, Foutse Khomh, Thomas Reid, and Ciro Guida, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022*

The following publication was produced during my Ph.D's degree studies but is not directly related to the content of the thesis:

- DiverGet: A Search-Based Software Testing Approach for Deep Neural Network Quantization Assessment, Ahmed Haj Yahmed, Houssem Ben Braiek, Foutse Khomh, Sonia Bouzidi, Rania Zaatour, in *Journal of Empirical Software Engineering (EMSE), 2022, Springer*

- Robustness Assessment of Hyperspectral Image CNNs using Metamorphic Testing, Rached Bouchoucha, Houssem Ben Braiek, Foutse Khomh, Sonia Bouzidi, Rania Zaatour, under review in *Information and Software Technology, 2022, Elsevier*

- Faults in Deep Reinforcement Learning Programs: A Taxonomy and A Detection Approach, Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek, in *Automated Software Engineering Journal (ASEJ), 2021, Springer*

- The Scent of Deep Learning Code: An Empirical Study, Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, Foutse Khomh, in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR), October 5 - 6, 2020, Yongsan-gu, Seoul, South Korea*

# REFERENCES


[1] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[2] J. M. M. C. Jacques Bughin, Jeongmin Seong and R. Joshi. (2018) Notes from the ai frontier: Modeling the impact of ai on the world economy, mckinsey global institute. [Online]. Available: https://www.mckinsey.com/featured-insights/artificial-intelligence/notes-from-the-ai-frontier-modeling-the-impact-of-ai-on-the-world-economy

[3] C. Ziegler, "A google self-driving car caused a crash for the first time," *https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report*, 2016.

[4] C. Metz, "After fatal uber crash, a self-driving start-up moves forward," *https://www.nytimes.com/2018/05/07/technology/uber-crash-autonomous-driveai.html*, 2018.

[5] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" in *Advances in Neural Information Processing Systems*, 2018, pp. 2483–2493.

[6] S. Li, J. Jiao, Y. Han, and T. Weissman, "Demystifying resnet," *arXiv preprint arXiv:1611.01186*, 2016.

[7] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[8] A. Karpathy, "Software 2.0," *https://medium.com/@karpathy/software-2-0-a64152b37c35*, 2018.

[9] S. Dargan, M. Kumar, M. R. Ayyagari, and G. Kumar, "A survey of deep learning and its applications: A new paradigm to machine learning," *Archives of Computational Methods in Engineering*, pp. 1–22, 2019.

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

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

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

[13] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning half-day tutorial," in *25th ACM Conference on Computer and Communications Security, CCS 2018*. Association for Computing Machinery, 2018, pp. 2154–2156.

[14] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[15] Z. Li, X. Ma, C. Xu, and C. Cao, "Structural coverage criteria for neural networks could be misleading," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019, pp. 89–92.

[16] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020.

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

[18] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 303–314.

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

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

[21] H. B. Braiek and F. Khomh, "Deepevolution: A search-based testing approach for deep neural networks," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 454–458.

[22] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 120–131.

[23] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1039–1049.

[24] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[26] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *arXiv preprint arXiv:2106.04554*, 2021.

[27] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Advances in Neural Information Processing Systems*, 2018, pp. 6389–6399.

[28] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," in *Advances in Neural Information Processing Systems*, 2018, pp. 7694–7705.

[29] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[30] P. D. Jing Zhi Loh and S. U. Azeem. (2020) Visualization: How to visualize, monitor and debug neural network learning. [Online]. Available: https://deeplearning4j.konduit.ai/tuning-and-training/visualization

[31] C. Garbin, X. Zhu, and O. Marques, "Dropout vs. batch normalization: an empirical study of their impact to deep learning," *Multimedia Tools and Applications*, pp. 1–39, 2020.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[33] P. Baldi and P. J. Sadowski, "Understanding dropout," in *Advances in neural information processing systems*, 2013, pp. 2814–2822.

[34] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[35] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron *et al.*, "Theano: Deep learning on gpus with python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, vol. 3, no. 0. Citeseer, 2011.

[36] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

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

[38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[39] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[40] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM'81 Conference*. ACM, 1981, pp. 254–257.

[41] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of

Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep., 1998.

[42] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[43] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational property-based testing," in *International Conference on Interactive Theorem Proving*. Springer, 2015, pp. 325–343.

[44] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[45] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[46] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 562–566.

[47] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[48] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, no. 2, pp. 239–287, 2009.

[49] S. Bhambri, S. Muku, A. Tulasi, and A. B. Buduru, "A survey of black-box adversarial attacks on computer vision models," *arXiv preprint arXiv:1912.01667*, 2019.

[50] R. Mosli, M. Wright, B. Yuan, and Y. Pan, "They might not be giants crafting black-box adversarial examples using particle swarm optimization," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 439–459.

[51] M. Alzantot, Y. Sharma, S. Chakraborty, H. Zhang, C.-J. Hsieh, and M. B. Srivastava, "Genattack: Practical black-box attacks with gradient-free optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1111–1119.

[52] J. Chen, M. Su, S. Shen, H. Xiong, and H. Zheng, "Poba-ga: Perturbation optimized black-box adversarial attacks via genetic algorithm," *Computers & Security*, vol. 85, pp. 89–106, 2019.

[53] X. Wang, H. Jin, and K. He, "Natural language adversarial attacks and defenses in word level," *arXiv preprint arXiv:1909.06723*, 2019.

[54] Y. Zang, F. Qi, C. Yang, Z. Liu, M. Zhang, Q. Liu, and M. Sun, "Word-level textual adversarial attacking as combinatorial optimization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 6066–6080.

[55] X. Du, J. Yu, Z. Yi, S. Li, J. Ma, Y. Tan, and Q. Wu, "A hybrid adversarial attack for different application scenarios," *Applied Sciences*, vol. 10, no. 10, p. 3559, 2020.

[56] T. Du, S. Ji, J. Li, Q. Gu, T. Wang, and R. Beyah, "Sirenattack: Generating adversarial audio for end-to-end acoustic systems," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 357–369.

[57] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on.* IEEE, 1995, pp. 39–43.

[58] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[59] S. Picek and M. Golub, "Comparison of a crossover operator in binary-coded genetic algorithms," *WSEAS transactions on computers*, vol. 9, no. 9, pp. 1064–1073, 2010.

[60] H. Chen, D. Dean, and D. A. Wagner, "Model checking one million lines of c code." in *NDSS*, vol. 4, 2004, pp. 171–185.

[61] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 202–216.

[62] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[63] E. Pira, V. Rafe, and A. Nikanjam, "Deadlock detection in complex software systems specified through graph transformation using bayesian optimization algorithm," *Journal of Systems and Software*, vol. 131, pp. 181–200, 2017.

[64] R. Heckel, "Graph transformation in a nutshell," *Electronic notes in theoretical computer science*, vol. 148, no. 1, pp. 187–198, 2006.

[65] S. Ciraci, P. van den Broek, and M. Aksit, "Graph-based verification of static program constraints," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 2265–2272.

[66] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020.

[67] R. G. Iyer, Y. Sun, W. Wang, and J. Gottschlich, "Software language comprehension using a program-derived semantic graph," in *NeurIPS 2020 Computer-Assisted Programming Workshop*, 2020.

[68] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1711.00740

[69] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[70] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, "A rotation and a translation suffice: Fooling CNNs with simple transformations," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1712.02779

[71] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[72] J. Gilmer, R. P. Adams, I. Goodfellow, D. Andersen, and G. E. Dahl, "Motivating the rules of the game for adversarial example research," *arXiv preprint arXiv:1807.06732*, 2018.

[73] S. S. Banerjee, S. Jha, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Hands off the wheel in autonomous vehicles?" *A systems.*

[74] M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz, "Invariant risk minimization," *arXiv preprint arXiv:1907.02893*, 2019.

[75] B. L. Sturm, "A simple method to determine if a music information retrieval system is a "horse"," *IEEE Transactions on Multimedia*, vol. 16, no. 6, pp. 1636–1644, 2014.

[76] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *CVPR 2011*. IEEE, 2011, pp. 1521–1528.

[77] A. Jabri, A. Joulin, and L. Van Der Maaten, "Revisiting visual question answering baselines," in *European conference on computer vision*. Springer, 2016, pp. 727–739.

[78] R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut learning in deep neural networks," *Nature Machine Intelligence*, vol. 2, no. 11, pp. 665–673, 2020.

[79] A. Mitani, A. Karthikesalingam, A. Beutel, A. N. D'Amour, A. Montanari, B. Alipanahi, B. Adlam, C. Chen, C. Nielsen, C. McLean, D. Sculley, D. Moldovan, D. Mincu, F. Hormozdiari, G. Jerfel, H. Suresh, J. Eisenstein, J. Schrouff, J. Deaton, K. Heller, K. Webster, K. Ramasamy, M. Lučić, M. G. Seneviratne, M. D. Hoffman, M. Vladymyrov, N. Houlsby, R. Raman, R. A. Sayres, S. Sequeira, S. Hou, S. Yadlowsky, T. Yun, T. Osborne, V. Veitch, V. Natarajan, X. Zhai, X. Wang, Y. Ma, and Z. Nado, "Underspecification presents challenges for credibility in modern machine learning," *Journal of Machine Learning Research*, 2020. [Online]. Available: https://arxiv.org/abs/2011.03395

[80] R. Caruana, Y. Lou, J. Gehrke, P. Koch, M. Sturm, and N. Elhadad, "Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, 2015, pp. 1721–1730.

[81] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," *Advances in neural information processing systems*, vol. 32, 2019.

[82] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua, "Quantization networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7308–7316.

[83] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[84] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[85] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized out-of-distribution detection: A survey," *arXiv preprint arXiv:2110.11334*, 2021.

[86] T. Li, A. Beirami, M. Sanjabi, and V. Smith, "Tilted empirical risk minimization," *arXiv preprint arXiv:2007.01162*, 2020.

[87] J. C. Duchi and H. Namkoong, "Learning models with uniform performance via distributionally robust optimization," *The Annals of Statistics*, vol. 49, no. 3, pp. 1378–1406, 2021.

[88] J. Liu, Z. Hu, P. Cui, B. Li, and Z. Shen, "Heterogeneous risk minimization," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6804–6814.

[89] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[90] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[91] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications." in *SEKE*, vol. 167, 2007.

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

[93] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "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*, 2018, pp. 118–128.

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

[95] J. Zhang, Y. Wang, P. Molino, L. Li, and D. S. Ebert, "Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 364–373, 2018.

[96] M. Liu, S. Liu, H. Su, K. Cao, and J. Zhu, "Analyzing the noise robustness of deep neural networks," in *2018 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, 2018, pp. 60–71.

[97] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. P. Chau, "A cti v is: Visual exploration of industry-scale deep neural network models," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 88–97, 2017.

[98] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu, "Towards better analysis of deep convolutional neural networks," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 91–100, 2016.

[99] N. Pezzotti, T. Höllt, J. Van Gemert, B. P. Lelieveldt, E. Eisemann, and A. Vilanova, "Deepeyes: Progressive visual analytics for designing deep neural networks," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 98–108, 2017.

[100] H. B. Braiek and F. Khomh, "Tfcheck: A tensorflow library for detecting training issues in neural network programs," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2019, pp. 426–433.

[101] E. Schoop, F. Huang, and B. Hartmann, "Scram: Simple checks for realtime analysis of model training for non-expert ml programmers," in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–10.

[102] ——, "Umlaut: Debugging deep learning programs using program structure and model behavior," 2021.

[103] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[104] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2574–2582.

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

[106] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

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

[108] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[109] N. Narodytska and S. P. Kasiviswanathan, "Simple black-box adversarial attacks on deep neural networks." in *CVPR Workshops*, vol. 2, 2017.

[110] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM workshop on artificial intelligence and security*, 2017, pp. 15–26.

[111] A. N. Bhagoji, W. He, B. Li, and D. Song, "Practical black-box attacks on deep neural networks using efficient query mechanisms," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 154–169.

[112] J. C. Spall, *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons, 2005, vol. 65.

[113] S. Moon, G. An, and H. O. Song, "Parsimonious black-box adversarial attacks via efficient combinatorial optimization," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4636–4645.

[114] W. Shen, J. Wan, and Z. Chen, "Munn: Mutation analysis of neural networks," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 108–115.

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

[116] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," 2001.

[117] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang, "Combinatorial testing for deep learning systems," *arXiv preprint arXiv:1806.07723*, 2018.

[118] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[119] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

[120] S. Bulusu, B. Kailkhura, B. Li, P. K. Varshney, and D. Song, "Anomalous example detection in deep learning: A survey," *IEEE Access*, vol. 8, pp. 132 330–132 347, 2020.

[121] M. Salehi, H. Mirzaei, D. Hendrycks, Y. Li, M. H. Rohban, and M. Sabokrou, "A unified survey on anomaly, novelty, open-set, and out-of-distribution detection: Solutions and future challenges," *arXiv preprint arXiv:2110.14051*, 2021.

[122] H. Choi, E. Jang, and A. A. Alemi, "Waic, but why? generative ensembles for robust anomaly detection," *arXiv preprint arXiv:1810.01392*, 2018.

[123] E. Nalisnick, A. Matsukawa, Y. W. Teh, D. Gorur, and B. Lakshminarayanan, "Do deep generative models know what they don't know?" *arXiv preprint arXiv:1810.09136*, 2018.

[124] S. Pidhorskyi, R. Almohsen, and G. Doretto, "Generative probabilistic novelty detection with adversarial autoencoders," *Advances in neural information processing systems*, vol. 31, 2018.

[125] D. Hendrycks, M. Mazeika, and T. Dietterich, "Deep anomaly detection with outlier exposure," *arXiv preprint arXiv:1812.04606*, 2018.

[126] V. Kuleshov, N. Fenner, and S. Ermon, "Accurate uncertainties for deep learning using calibrated regression," in *International conference on machine learning*. PMLR, 2018, pp. 2796–2804.

[127] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural network," in *International conference on machine learning*. PMLR, 2015, pp. 1613–1622.

[128] Y. Gal, J. Hron, and A. Kendall, "Concrete dropout," *Advances in neural information processing systems*, vol. 30, 2017.

[129] D. P. Kingma, T. Salimans, and M. Welling, "Variational dropout and the local reparameterization trick," *Advances in neural information processing systems*, vol. 28, 2015.

[130] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?" *Advances in neural information processing systems*, vol. 30, 2017.

[131] L. Mi, H. Wang, Y. Tian, H. He, and N. N. Shavit, "Training-free uncertainty estimation for dense regression: Sensitivity as a surrogate," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, 2022, pp. 10 042–10 050.

[132] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek, "Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift," *Advances in neural information processing systems*, vol. 32, 2019.

[133] E. J. Santana, R. P. Silva, B. B. Zarpelão, and S. Barbon Junior, "Detecting and mitigating adversarial examples in regression tasks: A photovoltaic power generation forecasting case study," *Information*, vol. 12, no. 10, p. 394, 2021.

[134] C. Zhang. (2018) How to choose last-layer activation and loss function. [Online]. Available: https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/

[135] L. N. Smith and N. Topin, "Deep convolutional neural network design patterns," in *International Conference on Learning Representations (ICLR)*, 2017.

[136] S. H. Hasanpour, M. Rouhani, M. Fayyaz, M. Sabokrou, and E. Adeli, "Towards principled design of deep convolutional networks: Introducing simpnet," *arXiv preprint arXiv:1802.06205*, 2018.

[137] T. Hartmann, A. Moawad, C. Schockaert, F. Fouquet, and Y. Le Traon, "Meta-modelling meta-learning," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2019, pp. 300–305.

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

[139] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[140] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," in *International Conference on Computational Sciences and Technology*, 2021.

[141] D. Mishkin, N. Sergievskiy, and J. Matas, "Systematic evaluation of convolution neural network advances on the imagenet," *Computer Vision and Image Understanding*, vol. 161, pp. 11–19, 2017.

[142] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[143] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

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

[145] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 2682–2690.

[146] D. Han, J. Kim, and J. Kim, "Deep pyramidal residual networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5927–5935.

[147] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," 2015.

[148] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *International conference on artificial neural networks*. Springer, 2010, pp. 92–101.

[149] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[150] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[151] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[152] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[153] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 5353–5360.

[154] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[155] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, vol. 25, 2012, pp. 1097–1105.

[156] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[157] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "Densenet: Implementing efficient convnet descriptor pyramids," *arXiv preprint arXiv:1404.1869*, 2014.

[158] (2020) Replication package and source code of . Https://github.com/neuralint/neuralint.

[159] R. M. Soley. (2013) How to deliver resilient, secure, efficient, and easily changed it systems in line with cisq recommendations. [Online]. Available: https://web.archive.org/web/20131228132152/http://www.omg.org/CISQ_compliant_IT_Systemsv.4-3.pdf

[160] A. Rensink, "The groove simulator: A tool for state space generation," in *International Workshop on Applications of Graph Transformations with Industrial Relevance*. Springer, 2003, pp. 479–485.

[161] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using groove," *International journal on software tools for technology transfer*, vol. 14, no. 1, pp. 15–40, 2012.

[162] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 1135–1146.

[163] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, p. 5, 2015.

[164] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[165] (2020) Tensorflow-based lenet. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/slim/nets/lenet.py

[166] (2020) Keras-based vgg16. Https://github.com/keras-team/keras-applications/blob/master/keras_applications/vgg16.py.

[167] https://github.com/taashi-s/UNet_Keras/commit/b1b6d93, 2018, accessed: 2021-02-05.

[168] https://github.com/dishen12/keras_frcnn/commit/38413c6, 2017, accessed: 2021-02-05.

[169] https://github.com/keras-team/keras-applications/commit/05ff470, 2018, accessed: 2021-02-05.

[170] https://github.com/mateusz93/Car-recognition/commit/94b36ea, 2018, accessed: 2021-02-05.

[171] https://github.com/yumatsuoka/comp_DNNfw/commit/30e0973, 2017, accessed: 2021-02-05.

[172] https://github.com/tf-encrypted/tf-encrypted/issues/248 (network B), 2018, accessed: 2021-02-05.

[173] https://github.com/tf-encrypted/tf-encrypted/issues/248 (network C), 2018, accessed: 2021-02-05.

[174] https://github.com/katyprogrammer/regularization-experiment/commit/b93dd636, 2016, accessed: 2021-02-05.

[175] F. M. Hohman, M. Kahng, R. Pienta, and D. H. Chau, "Visual analytics in deep learning: An interrogative survey for the next frontiers," *IEEE transactions on visualization and computer graphics*, 2018.

[176] S. Ivanov. (2017) 37 reasons why your neural network is not working. [Online]. Available: https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

[177] C. Shao. (2019) Checklist for debugging neural networks. [Online]. Available: https://towardsdatascience.com/checklist-for-debugging-neural-networks-d8b2a9434f21

[178] D. Bansal. (2020) Pitfalls of batch norm in tensorflow and sanity checks for training networks. [Online]. Available: https://towardsdatascience.com/pitfalls-of-batch-norm-in-tensorflow-and-sanity-checks-for-training-networks-e86c207548c8

[179] Deeplearning4j. (2019) Troubleshooting neural net training. [Online]. Available: https://deeplearning4j.konduit.ai/tuning-and-training/troubleshooting-training

[180] A. Karpathy. (2020) A recipe for training neural networks. [Online]. Available: http://karpathy.github.io/2019/04/25/recipe/

[181] R. D. community. (2018) My neural network isn't working! what should i do? [Online]. Available: https://www.reddit.com/r/MachineLearning/comments/6xvnwo/d_my_neural_network_isnt_working_what_should_i_do/

[182] A. Karpathy. (2018) Most common neural net mistakes (tweet). [Online]. Available: https://twitter.com/karpathy/status/1013244313327681536?lang=en

[183] R. Grosse, "Lecture 15: Exploding and vanishing gradients," *University of Toronto Computer Science*, 2017.

[184] J. Wang, L. Perez *et al.*, "The effectiveness of data augmentation in image classification using deep learning," *Convolutional Neural Networks Vis. Recognit*, vol. 11, pp. 1–8, 2017.

[185] B. Van Rooyen and R. C. Williamson, "A theory of learning with corrupted labels." *Journal of Machine Learning Research*, vol. 18, pp. 228–1, 2017.

[186] C. Roberts. (2018) How to unit test machine learning code. [Online]. Available: https://medium.com/@keeper6928/how-to-unit-test-machine-learning-code-57cf6fd81765

[187] U. Evci, "Detecting dead weights and units in neural networks," Master's thesis, New York University, New York, 2018.

[188] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[189] Y. Gal and Z. Ghahramani, "Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference," 2016.

[190] H. Bae, J. Jang, D. Jung, H. Jang, H. Ha, and S. Yoon, "Privacy and security issues in deep learning: A survey," *IEEE Access*, vol. 9, pp. 4566–4593, 2021.

[191] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," in *Workshop on Formal Verification of Autonomous Vehicles, International Conference on Integrated Formal Methods (iFM)*, 2017.

[192] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "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*. ACM, 2019, pp. 625–635.

[193] MIT. (2020) Mit deep learning lectures. [Online]. Available: https://deeplearning.mit.edu/

[194] ——. (2020) Cs231n: Convolutional neural networks for visual recognition. [Online]. Available: https://cs231n.github.io/

[195] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 1135–1146.

[196] Google. (2020) Google machine learning crash course. [Online]. Available: https://developers.google.com/machine-learning/crash-course

[197] A. Ng, K. Katanforoosh, and Y. B. Mourri. (2020) Specialization in deep learning. [Online]. Available: https://www.coursera.org/specializations/deep-learning

[198] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[199] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[200] T. J. Archdeacon, *Correlation and regression analysis: a historian's guide*. Univ of Wisconsin Press, 1994.

[201] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[202] G. E. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 599–619.

[203] A. Karpathy. (2018) Neural networks part 3: Learning and evaluation. [Online]. Available: http://cs231n.github.io/neural-networks-3/

[204] D. H. Park, C. M. Ho, Y. Chang, and H. Zhang, "Gradient-coherent strong regularization for deep neural networks," *arXiv preprint arXiv:1811.08056*, 2018.

[205] L. Bottou. (2015) Multilayer neural networks. [Online]. Available: http://videolectures.net/site/normal_dl/tag=983658/deeplearning2015_bottou_neural_networks_01.pdf

[206] A. Rakitianskaia and A. Engelbrecht, "Measuring saturation in neural networks," in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1423–1430.

[207] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *arXiv preprint arXiv:1903.06733*, 2019.

[208] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," in *International Conference on Machine Learning*, 2019.

[209] H. B. Braiek, F. Khomh, and B. Adams, "The open-closed principle of modern machine learning frameworks," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 353–363.

[210] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "Tensorflow debugger: Debugging dataflow graphs for machine learning," 2016.

[211] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[212] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[213] P. S. Foundation. (2013) Mutpy 0.4.0. [Online]. Available: https://pypi.python.org/pypi/

[214] Google. (2017) Basic regression: Predict fuel efficiency. [Online]. Available: https://www.tensorflow.org/tutorials/keras/regression

[215] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[216] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[217] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *http://www.cs.toronto.edu/kriz/cifar.html*, 2014.

[218] N. Rauschmayr, V. Kumar, R. Huilgol, A. Olgiati, S. Bhattacharjee, N. Harish, V. Kannan, A. Lele, A. Acharya, J. Nielsen *et al.*, "Amazon sagemaker debugger: A system for real-time insights into machine learning model training," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[219] A. Team. (2021) List of debugger built-in rules. [Online]. Available: https://docs.aws.amazon.com/sagemaker/latest/dg/debugger-built-in-rules.html

[220] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[221] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*. PMLR, 2013, pp. 1310–1318.

[222] K. Roth, A. Lucchi, S. Nowozin, and T. Hofmann, "Stabilizing training of generative adversarial networks through regularization," *Advances in neural information processing systems*, vol. 30, 2017.

[223] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *arXiv preprint arXiv:1701.04862*, 2017.

[224] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," in *International Conference on Learning Representations*, 2018.

[225] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-100 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[226] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[227] M. G. de Pinto, M. Polignano, P. Lops, and G. Semeraro, "Emotions understanding model from spoken language using deep neural networks and mel-frequency cepstral

coefficients," in *2020 IEEE conference on evolving and adaptive intelligent systems (EAIS)*. IEEE, 2020, pp. 1–5.

[228] A. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, 2011, pp. 142–150.

[229] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[230] M. Zhao, J.-S. Pan, and S.-T. Chen, "Optimal snr of audio watermarking by wavelet and compact pso methods." *J. Inf. Hiding Multim. Signal Process.*, vol. 6, no. 5, pp. 833–846, 2015.

[231] Z. Su, G. Zhang, F. Yue, L. Chang, J. Jiang, and X. Yao, "Snr-constrained heuristics for optimizing the scaling parameter of robust audio watermarking," *IEEE Transactions on Multimedia*, vol. 20, no. 10, pp. 2631–2644, 2018.

[232] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 2890–2896. [Online]. Available: https://aclanthology.org/D18-1316

[233] S. Kullback, "Letter to the editor: The kullback-leibler distance," 1987.

[234] T. M. Cover and J. A. Thomas, *Elements of information theory*, ser. Wiley series in telecommunications. New York: Wiley, 1991.

[235] Y.-C. Ho and D. L. Pepyne, "Simple explanation of the no-free-lunch theorem and its implications," *Journal of optimization theory and applications*, vol. 115, no. 3, pp. 549–570, 2002.

[236] X.-S. Yang, "Firefly algorithm, stochastic test functions and design optimisation," *arXiv preprint arXiv:1003.1409*, 2010.

[237] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in engineering software*, vol. 69, pp. 46–61, 2014.

[238] S. Mirjalili, "Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm," *Knowledge-Based Systems*, vol. 89, pp. 228–249, 2015.

[239] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016.

[240] S. Mirjalili, S. M. Mirjalili, and A. Hatamlou, "Multi-verse optimizer: a nature-inspired algorithm for global optimization," *Neural Computing and Applications*, vol. 27, no. 2, pp. 495–513, 2016.

[241] S. Mirjalili, A. H. Gandomi, S. Z. Mirjalili, S. Saremi, H. Faris, and S. M. Mirjalili, "Salp swarm algorithm: A bio-inspired optimizer for engineering design problems," *Advances in Engineering Software*, vol. 114, pp. 163–191, 2017.

[242] S. K. Joshi and J. C. Bansal, "Parameter tuning for meta-heuristics," *Knowledge-Based Systems*, vol. 189, p. 105094, 2020.

[243] M. A. Hossan, S. Memon, and M. A. Gregory, "A novel approach for mfcc feature extraction," in *2010 4th International Conference on Signal Processing and Communication Systems*. IEEE, 2010, pp. 1–5.

[244] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[245] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[246] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[247] Y. A. Ioannou, "Structural priors in deep neural networks," Ph.D. dissertation, University of Cambridge, 2018.

[248] H. Ren, R. Stewart, J. Song, V. Kuleshov, and S. Ermon, "Learning with weak supervision from physics and data-driven constraints," *AI Magazine*, vol. 39, no. 1, pp. 27–38, 2018.

[249] R. Stewart and S. Ermon, "Label-free supervision of neural networks with physics and domain knowledge," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[250] N. Muralidhar, J. Bu, Z. Cao, L. He, N. Ramakrishnan, D. Tafti, and A. Karpatne, "Physics-guided design and learning of neural networks for predicting drag force on particle suspensions in moving fluids," *arXiv preprint arXiv:1911.04240*, 2019.

[251] X. Wang, Y. Zhao, and F. Pourpanah, "Recent advances in deep learning," 2020.

[252] A. T. Nguyen and E. Raff, "Adversarial attacks, regression, and numerical stability regularization," *arXiv preprint arXiv:1812.02885*, 2018.

[253] A. Karpatne, G. Atluri, J. H. Faghmous, M. Steinbach, A. Banerjee, A. Ganguly, S. Shekhar, N. Samatova, and V. Kumar, "Theory-guided data science: A new paradigm for scientific discovery from data," *IEEE Transactions on knowledge and data engineering*, vol. 29, no. 10, pp. 2318–2331, 2017.

[254] J. Ling, A. Kurzawski, and J. Templeton, "Reynolds averaged turbulence modelling using deep neural networks with embedded invariance," *Journal of Fluid Mechanics*, vol. 807, pp. 155–166, 2016.

[255] S. Seo and Y. Liu, "Differentiable physics-informed graph networks," *arXiv preprint arXiv:1902.02950*, 2019.

[256] J. Z. Leibo, Q. Liao, F. Anselmi, W. A. Freiwald, and T. Poggio, "View-tolerant face recognition and hebbian learning imply mirror-symmetric neural tuning to head orientation," *Current Biology*, vol. 27, no. 1, pp. 62–67, 2017.

[257] M. Marini, R. Paoli, F. Grasso, J. Periaux, and J.-A. Desideri, "Verification and validation in computational fluid dynamics: the flownet database experience," *JSME International Journal Series B Fluids and Thermal Engineering*, vol. 45, no. 1, pp. 15–22, 2002.

[258] P. J. Roache, *Verification and validation in computational science and engineering*. Hermosa Albuquerque, NM, 1998, vol. 895.

[259] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?" *Neural computation*, vol. 16, no. 5, pp. 1063–1076, 2004.

[260] C. of Federal Regulations. Cfr: Induction system icing. [Online]. Available: https://www.law.cornell.edu/cfr/text/14/33.68

[261] M. Eshelby, *Aircraft performance: Theory and practice*. American Institute of Aeronautics and Astronautics, Inc., 2000.

[262] D. W. Hahn and M. N. Özisik, *Heat conduction*. John Wiley & Sons, 2012.

[263] F. P. Incropera, D. P. DeWitt, T. L. Bergman, A. S. Lavine *et al.*, *Fundamentals of heat and mass transfer*. Wiley New York, 1996, vol. 6.

[264] V. Sarge, M. Andersch, L. Fabel, P. Micikevicius, and J. Tran. (2019) Tips for optimizing gpu performance using tensor cores. [Online]. Available: https://developer.nvidia.com/blog/optimizing-gpu-performance-tensor-cores/

[265] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *International Conference on Neural Information Processing Systems*, 2019.

[266] G. Sun and S. Wang, "A review of the artificial neural network surrogate modeling in aerodynamic design," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 233, no. 16, pp. 5863–5872, 2019.

[267] D. Cao and G.-C. Bai, "Dnn-based surrogate modeling-based feasible performance reliability design methodology for aircraft engine," *IEEE Access*, vol. 8, pp. 229 201–229 218, 2020.

[268] Q. Du, T. Liu, L. Yang, L. Li, D. Zhang, and Y. Xie, "Airfoil design and surrogate modeling for performance prediction based on deep learning method," *Physics of Fluids*, vol. 34, no. 1, p. 015111, 2022.

[269] M. Ahmed and N. Qin, "Surrogate-based aerodynamic design optimization: Use of surrogates in aerodynamic design optimization," in *International Conference on Aerospace Sciences and Aviation Technology*, vol. 13, no. AEROSPACE SCIENCES & AVIATION TECHNOLOGY, ASAT-13, May 26–28, 2009. The Military Technical College, 2009, pp. 1–26.

[270] A. Sobester, A. Forrester, and A. Keane, *Engineering design via surrogate modelling: a practical guide*. John Wiley & Sons, 2008.

[271] J. Sobieszczanski-Sobieski and R. T. Haftka, "Multidisciplinary aerospace design optimization: survey of recent developments," *Structural optimization*, vol. 14, no. 1, pp. 1–23, 1997.

[272] T. W. Simpson, T. M. Mauery, J. J. Korte, and F. Mistree, "Kriging models for global approximation in simulation-based multidisciplinary design optimization," *AIAA journal*, vol. 39, no. 12, pp. 2233–2241, 2001.

[273] Y. Kuya, K. Takeda, X. Zhang, and A. I. Forrester, "Multifidelity surrogate modeling of experimental and computational aerodynamic data sets," *AIAA journal*, vol. 49, no. 2, pp. 289–298, 2011.

[274] S. G. Kontogiannis, A. M. Savill, and T. Kipouros, "A multi-objective multi-fidelity framework for global optimization," in *58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2017, p. 0136.

[275] E. Iuliano and D. Quagliarella, "Proper orthogonal decomposition, surrogate modelling and evolutionary optimization in aerodynamic design," *Computers & Fluids*, vol. 84, pp. 327–350, 2013.

[276] G. Pagliuca and S. Timme, "Model reduction for flight dynamics simulations using computational fluid dynamics," *Aerospace Science and Technology*, vol. 69, pp. 15–26, 2017.

[277] D. J. Linse and R. F. Stengel, "Identification of aerodynamic coefficients using computational neural networks," *Journal of Guidance, Control, and Dynamics*, vol. 16, no. 6, pp. 1018–1025, 1993.

[278] M. M. Rai and N. K. Madavan, "Aerodynamic design using neural networks," *AIAA journal*, vol. 38, no. 1, pp. 173–182, 2000.

[279] D. Rajaram, T. G. Puranik, A. Renganathan, W. J. Sung, O. J. Pinon-Fischer, D. N. Mavris, and A. Ramamurthy, "Deep gaussian process enabled surrogate models for aerodynamic flows," in *AIAA Scitech 2020 Forum*, 2020, p. 1640.

[280] L. Zhao, K. Choi, and I. Lee, "Metamodeling method using dynamic kriging for design optimization," *AIAA journal*, vol. 49, no. 9, pp. 2034–2046, 2011.

[281] T. Kipouros, M. Molinari, W. N. Dawes, G. T. Parks, M. Savill, and K. W. Jenkins, "An investigation of the potential for enhancing the computational turbomachinery design cycle using surrogate models and high performance parallelisation," in *Turbo Expo: Power for Land, Sea, and Air*, vol. 47950, 2007, pp. 1415–1424.

[282] G. Pagliuca, T. Kipouros, and M. Savill, "Surrogate modelling for wing planform multidisciplinary optimisation using model-based engineering," *International Journal of Aerospace Engineering*, vol. 2019, 2019.

[283] R. Novak, Y. Bahri, D. A. Abolafia, J. Pennington, and J. Sohl-Dickstein, "Sensitivity and generalization in neural networks: an empirical study," *arXiv preprint arXiv:1802.08760*, 2018.

[284] M. Rosca, T. Weber, A. Gretton, and S. Mohamed, "A case for new neural network smoothness constraints," 2020.

[285] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[286] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[287] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: removing noise by adding noise," *arXiv preprint arXiv:1706.03825*, 2017.

[288] G. Kovács, "An empirical comparison and evaluation of minority oversampling techniques on a large number of imbalanced datasets," *Applied Soft Computing*, vol. 83, p. 105662, 2019.

[289] J. D. Anderson, *Aircraft performance and design*. WCB/McGraw-Hill Boston, 1999, vol. 1.

[290] W.-L. Loh, "On latin hypercube sampling," *The annals of statistics*, vol. 24, no. 5, pp. 2058–2080, 1996.

[291] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, 1994.

[292] R. E. Wright, "Logistic regression." 1995.

[293] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[294] G. Biau and E. Scornet, "A random forest guided tour," *Test*, vol. 25, no. 2, pp. 197–227, 2016.

[295] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, p. 21, 2013.

[296] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[297] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-smote: a new over-sampling method in imbalanced data sets learning," in *International conference on intelligent computing*. Springer, 2005, pp. 878–887.

[298] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*. IEEE, 2008, pp. 1322–1328.

[299] Y. Tang, Y.-Q. Zhang, N. V. Chawla, and S. Krasser, "Svms modeling for highly imbalanced classification," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 1, pp. 281–288, 2008.

[300] Wendl, *Scattered Data Approximation (Cambridge Monographs on Applied and Computational Mathematics; 17)*. Cambridge University Press, 2004.

[301] D. G. Krige, "A statistical approach to some basic mine valuation problems on the witwatersrand," *Journal of the Southern African Institute of Mining and Metallurgy*, vol. 52, no. 6, pp. 119–139, 1951.

[302] M. D. Buhmann, *Radial basis functions: theory and implementations*. Cambridge university press, 2003, vol. 12.

[303] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 233–240.

[304] GPy, "GPy: A gaussian process framework in python," http://github.com/SheffieldML/GPy, since 2012.

[305] Q. Zhang, Y. Zhao, W. Sun, C. Fang, Z. Wang, and L. Zhang, "Program repair: Automated vs. manual," *arXiv preprint arXiv:2203.05166*, 2022.

[306] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021.

[307] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A survey of deep active learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.