Titre: Title:	Who Tests the Testers? Assessing the Effectiveness and Trustworthiness of Deep Learning Model Testing Techniques
Auteur: Author:	Florian Tambon
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Tambon, F. (2024). Who Tests the Testers? Assessing the Effectiveness and Trustworthiness of Deep Learning Model Testing Techniques [Ph.D. thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/59454/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/59454/
Directeurs de recherche: Advisors:	Foutse Khomh, & Giuliano Antoniol
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Who Tests the Testers? Assessing the Effectiveness and Trustworthiness of Deep Learning Model Testing Techniques

FLORIAN TAMBON
Département de génie informatique et génie logiciel

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

Septembre 2024

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Who Tests the Testers? Assessing the Effectiveness and Trustworthiness of Deep Learning Model Testing Techniques

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

Michel DESMARAIS, président
Foutse KHOMH, membre et directeur de recherche
Antoniol GIULIANO, membre et codirecteur de recherche
Liam PAULL, membre
Lingming ZHANG, membre externe

DEDICATION

To my family

ACKNOWLEDGEMENTS

First, I want to express my gratitude to both my supervisors, Pr. Foutse Khomh and Pr. Giuliano Antoniol, for giving me the opportunity to do this PhD, for their advice which contributed to enhancing the quality of all my works and for their support and guidance throughout this journey. Secondly, I would like to thank Dr. Amin Nikanjam for his advice and precious suggestions during my PhD which helped better my work. I want to thank all my co-authors for their contributions, without which each work would not have been as good as they ended up being. My sincere thanks also go to the members of the DEEL project 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). Furthermore, I also would like to thank my committee members, Pr. Michel Desmarais, Pr. Liam Paull and Pr. Lingming Zhang, for evaluating this thesis. Finally, I thank my dearest friends and family for their support during this adventure.

RÉSUMÉ

L'arrivée des algorithmes d'apprentissage profond et leur intégration dans tous les domaines de la vie courante ont eu et continu d'avoir un impact important sur la société. Plus récemment, l'arrivée de l'Intelligence Artificielle générative à travers des technologies tel que ChatGPT ont accéléré ce processus. En parallèle, plusieurs recherches ont montré les limitations de ces systèmes au regard de leurs possibles défaillances, posant la question de comment faire en sorte de prévenir ces problèmes pour améliorer leur fiabilité.

Une manière établie de répondre à cette problématique consiste à tester ces systèmes, afin de détecter ces défaillances avant le déploiement de ces systèmes et trouver les fautes responsables et les réparer. À cet effet, plusieurs techniques de tests ont été développées au fil des années, en s'inspirant de techniques existantes dans le domaine du test logiciel ou en construisant de nouvelles méthodes adaptées à ces nouveaux algorithmes. Cependant, le développement de ces techniques a montré également que le nouveau paradigme apporté par l'apprentissage profond, i.e., le fait que la logique interne des modèles est "apprise" et non codée, a changé la donne. À travers cela, c'est la fiabilité et l'efficacité de ces méthodes de tests qui sont remises partiellement en question, ce qui compromet à son tour la fiabilité de ces algorithmes d'apprentissage profond et la confiance des utilisateurs.

C'est dans ce contexte que se situe cette thèse. Ce travail, organisé en huit chapitres, vise à adresser la question de la fiabilité des techniques de tests appliqués aux modèles d'apprentissage profond via le développement de quatre cadriciels pour améliorer la fiabilité et l'efficacité de techniques de tests. Chacun d'entre eux est orienté sur un aspect différent en termes des limites des techniques de tests et du sous-paradigme d'apprentissage profond concernés, peignant différentes possibilités d'améliorer ces techniques de tests. Ce travail commence par une introduction (Chapitre 1) contextualisant le problème avant de définir les connaissances préalables nécessaires à cette thèse (Chapitre 2). Par la suite, une revue de la littérature illustre les différentes problématiques et techniques existantes liées à ce travail (Chapitre 3). Cela donne lieu à quatre chapitres décrivant chacun un cadriciel particulier.

Le Chapitre 4 commence par explorer les techniques de tests visant la détection des fautes dans le code et spécifications des algorithmes par renforcement, en étendant le concept de tests par mutations à ce sous-paradigme de l'apprentissage profond. Le chapitre illustre les problématiques d'application de cette technique dans ce contexte et montre comment une solution peut être apportée via un cadriciel *RLMutation*.

Le Chapitre 5 prolonge le précèdent chapitre sur le sujet du test par mutation, en proposant

une généralisation du concept qui permet de combler les limitations de son utilisation rencontrées dans le chapitre précédent et dans l'état de l'art. Cette généralisation passe par la définition d'un autre cadriciel *Probabilistic Mutation Testing*, ou Test par Mutation Probabiliste, (PMT), utilisant les probabilités Bayésiennes. Le cadriciel est appliqué dans le cadre de l'apprentissage supervisé, dans un objectif de à la fois se comparer à l'état de l'art ainsi que pour montrer une application du test par mutations à un autre sous-paradigme, mais la définition est suffisamment générale pour qu'il puisse s'appliquer aux autres cadres.

Au contraire des chapitres précédents, le Chapitre 6 s'intéresse aux techniques qui touche à la détection des fautes dans un modèle précis. Plus particulièrement, le but de ce chapitre est de permettre de réduire le haut coût en temps d'application des techniques de générations de tests, en proposant un cadriciel de transfert nommé *Generated Inputs Sets Transfer*, ou Transfert pour Ensembles d'Entrées Générées, (GIST) qui propose une alternative à la simple ré-application des techniques de tests, via la définition du concept de transfert de tests, inspiré par le concept d'apprentissage par transfert.

Le Chapitre 7 touche également aux techniques appliquées aux fautes dans les modèles, et se concentre sur les problèmes de la quantification de la difficulté dans les tâches que ces modèles de langues effectuent via l'analyse de tâches de programmations. Les tâches évaluées à difficile sont de première importance car elles représentent les symptômes de fautes pour la plupart des modèles de langages quel que soit l'entrée donnée aux modèles et doivent donc être corrigées en priorité. Via la définition de *DiffEval*, le chapitre illustre les limites d'analyser les résultats sous un seul prisme en utilisant des métriques d'évaluation globale tel que l'exactitude, et montre l'utilité d'utiliser plusieurs entrées et modèles conjointement pour pouvoir estimer la difficulté d'une tâche. En particulier, l'identification de tâches difficiles pour ces modèles permet d'identifier des défaillances claires qui doivent faire l'objet d'amélioration. Le chapitre montre également les possibilités du cadriciel quant à la génération de nouvelles tâches à partir des tâches identifiées pour améliorer la procédure de test.

Finalement, le Chapitre 8 conclut la thèse en résumant les principaux points abordés dans les chapitres précédents, mettant en lumière les limites des méthodes proposées et les principales pistes d'amélioration futures.

ABSTRACT

The rise of Deep Learning models and their integration into daily life have impacted society, and the recent trend of Generative Artificial Intelligence models has boosted this process. However, in parallel to those developments, several studies have shown the limitations of such models, notably the dramatic failures they incur, raising the question of their trustworthiness.

One established way of dealing with this issue is to test those models so failures can be detected prior to deployment and the faults causing them to be identified and fixed. To that end, multiple testing techniques have been developed over the years, either adapted from traditional software testing or devised to adapt to those new models. However, the development of those techniques has shown that the new paradigm brought about by Deep Learning, that is, an inner logic "learned" and not coded, could impact the trustworthiness and effectiveness of the testing techniques itself, undermining the effort to foster users' trust in Deep Learning models.

This thesis takes place in that context and, organized in eight chapters, aims to deal with the trustworthiness of testing techniques applied to Deep Learning models by defining four frameworks for improving the trustworthiness and effectiveness of testing techniques in Deep Learning. Each framework is focused on a different aspect of the problem, both in terms of the limits of the testing techniques tackled and of the sub-paradigms of Deep Learning investigated, thus giving a comprehensive picture of the possible improvement. The thesis starts with an introduction (Chapter 1) to frame the problem the thesis is tackling and then provides prior knowledge of works dealt with in this work (Chapter 2). Then, a literature review (Chapter 3) is presented, describing related works and current problems related to the study. Finally, the following four chapters deal with the frameworks mentioned above.

Chapter 4 starts by exploring the testing techniques targeting faults in the code and specifications of Deep Reinforcement Learning algorithms, extending the concept of mutation testing to this particular sub-paradigm of Deep Learning. This chapter illustrates the application issues of mutation technique in Deep Reinforcement Learning and shows a possible solution through the proposed framework *RLMutation*.

Chapter 5 prolongs the topic of mutation testing by introducing a generalization of the concept to tackle limitations discussed in the previous chapter and present in state-of-the-art practices. The generalization relies on another framework named *Probabilistic Mutation Testing* (PMT), which leverages Bayesian probability. While the framework is applied to supervised learning to compare to state-of-the-art and tackle another sub-paradigm with

mutation testing, the definition is general enough to apply to a wide array of sub-paradigms and problems.

Contrary to previous chapters, Chapter 6 deals with testing techniques targeting faults within the models themselves. In particular, the aim is to reduce the high cost of applying test generation techniques through a transfer technique named *Generated Inputs Sets Transfer* (GIST). The framework does so by proposing an alternative to re-applying the techniques every time with the definition of the concept of test transfer inspired by transfer learning.

Chapter 7 similarly deals with techniques targeting faults within models themselves, yet focuses on quantifying the difficulty of coding tasks for those Large Language Models. Tasks labelled as hard are essential in that context, as they are symptoms of clear faults within most Large Language Models no matter the inputs and so should be patched up in priority. To do so, *DiffEval* framework is proposed, and the limitation of evaluating difficulty through the lens of one model or a limited set of inputs with global metrics such as accuracy is shown. Notably, the framework relies on multiple inputs and models to assess the task's difficulty properly. The chapter also indicates that DiffEval allows for generating new tasks from the hard tasks identified, further improving test techniques.

Finally, Chapter 8 concludes the thesis by summing up the main points tackled in previous chapters, mentioning the limitations of the proposed methods before highlighting the main future research avenues that could expand the work of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

DL	Deep Learning
AI	Artificial Intelligence
ST	Software Testing
RQ	Research Question
PMT	Probablistic Mutation Testing
MT	Mutation Testing
GIST	Generated Inputs Sets Transfer
DRL	Deep Reinforcement Learning
RL	Reinforcement Learning
HOM	Higher Order Mutation
FOM	First Order Mutation
DNN	Deep Neural Network
ML	Machine Learning
LLM	Large Language Model
ReLU	REctified Linear Unit
IOU	Intersection Over Union
BLEU	Bilingual Evaluation Understudy
AST	Abstract Synthax Tree
GPU	Graphics Processing Unit
API	Application programming interface
RMSE	Root Mean Square Error
MSE	Mean Square Error
METEOR	Metric for Evaluation of Translation with Explicit ORdering
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
BERT	Bidirectional Encoder Representations from Transformers
NC	Neuron Coverage
KMNC	K-Multisection Neuron Coverage
MC/DC	Modified Condition/Decision Coverage
GAN	Generative Adversarial Network
AVG	Average (criterion used in Chapter 4)
R	Reward (criterion used in Chapter 4)
DtR	Distance to Reward (criterion used in Chapter 4)
RN	Reward Noise (mutation used in Chapter 4)

M	Mangled (mutation used in Chapter 4)
Ra	Random (mutation used in Chapter 4)
Re	Repeat (mutation used in Chapter 4)
NDF	No Discount Factor (mutation used in Chapter 4)
MTS	Missing Terminal State (mutation used in Chapter 4)
NR	No Reverse (mutation used in Chapter 4)
MSU	Missing State Update (mutation used in Chapter 4)
ILF	Incorrect Loss Function (mutation used in Chapter 4)
PAC	Policy Activation Change (mutation used in Chapter 4)
POC	Policy Optimizer Change (mutation used in Chapter 4)
SGD	Stochastic Gradient Descent
DQN	Deep Q-Learning (RL algorithm used in Chapter 4)
A2C	Advantage Actor-Critics (RL algorithm used in Chapter 4)
PPO	Proximal Policy Optimization (RL algorithm used in Chapter 4)
NS	Non Subsuming (type of HOM used in Chapter 4)
WSC	Weakly Subsuming Coupled (type of HOM used in Chapter 4)
WSD	Weakly Subsuming Decoupled (type of HOM used in Chapter 4)
SSC	Strongly Subsuming Coupled (type of HOM used in Chapter 4)
MAP	Maximum A Posteriori (point estimate in Chapter 5)
MMSE	Minimum Mean Square Error (point estimate in Chapter 5)
CI	Credible Interval
MCE	Monte-Carlo Error
MN	MNIST Model (model in Chapter 5)
MR	Movie Recommender Model (model in Chapter 5)
UE	UnityEyes Model (model in Chapter 5)
TCL	Change Label (mutation used in Chapter 5)
TRD	Delete Training Data (mutation used in Chapter 5)
WCI	Change Weights Initialisation (mutation used in Chapter 5)
ACH	Change Activation Function (mutation used in Chapter 5)
TUD	Unbalance Training Data (mutation used in Chapter 5)
LCH	Change Loss Function (mutation used in Chapter 5)
OCH	Change Optimisation Function (mutation used in Chapter 5)
UMAP	Uniform Manifold Approximation and Projection (UMAP)
HDBSCAN	Hierachical Density-Based Spatial Clustering of Applications with Noise
DBCV	Density-Based Clustering Validation

NLP	Natural Language Processing
PWCCA	Projection Weighted Canonical Correlation Analysis (similarity distance in Chapter 6)
CKA	Centered Kernel Alignment (similarity distance in Chapter 6)
Ortho	Procrustes Orthogonal (similarity distance in Chapter 6)
Acc	Performance (similarity distance in Chapter 6)
Dis	Disagreement (similarity distance in Chapter 6)
Div	Divergence (similarity distance in Chapter 6)
OBF	Overall Best First (heuristic criterion in Chapter 6)
EBF	Each Best First (heuristic criterion in Chapter 6)

CHAPTER 1 INTRODUCTION

Deep Learning (DL) is drastically changing the way we interact with the world. We now use software applications powered by DL in critical aspects of our daily lives, from finance and energy to health and transportation. Moreover, the Artificial Intelligence (AI) market is predicted to reach $407 billion in revenues by 2027 [4]. In particular, the latest Generative AI revolution propelled by models such as ChatGPT [5] or Copilot [6] are playing a significant role in this transformation. Yet, at the same time, such systems can be prone to failures that can have drastic consequences if not appropriately addressed. For instance, car autopilot features can lead to fatal accidents and crashes [7]. As such, finding faults in DL models, i.e., the cause of those failures, is paramount.

A prominent way of detecting faults within DL models is by *testing* them, that is, feeding models with various scenarios and comparing the models' behaviour with an intended outcome. The objective is to identify inputs that cause the models to enter those failure modes, that is, inputs leading to a behaviour different than expected, to localize and fix the faults causing them. However, DL models function differently than traditional programs, as instead of having an inner logic coded by developers, the DL models "learns" this logic using available data through a stochastic learning algorithm. Thus, faults can occur in the code supporting the DL models, inside the data, or during the learning. On top of that, DL models can vary drastically in terms of architecture, learning paradigms or objectives depending on the particular paradigms of DL used. For instance, training DL models through supervised learning differs from training them with reinforcement learning. To that end, multiple testing techniques have been proposed, such as using metrics like Accuracy to evaluate DL models' response on a set of inputs [8], reusing traditional Software Testing (ST) approaches such as mutation testing [9] or fuzz testing [10], coming up with new coverage criteria [11], generating or handcrafting new inputs based on a target objective [12] etc.

An implicit assumption is that the testing approaches should be both effective and trustworthy in assessing DL models so users can, in turn, trust the DL models they are using. However, current testing approaches suffer from various limitations [13, 14] due to the nature of DL. These limitations impact models testing trustworthiness and efficiency. For instance, the inherent stochasticity of the learning/inference process [15, 16] (e.g., numerical instability, random initialization etc.), the ill-definition of some testing approach for certain DL paradigms [17], the high cost of applying those testing techniques [18] or the limited local assessment of the test in favour of a more global view. Such limitations emphasize the need

for more comprehensive frameworks to acknowledge and address these downsides.

1.1 Research questions

The objective of this thesis is to address the following high-level question:

How can we enhance the trustworthiness and effectiveness of testing techniques applied to Deep Learning models to foster users trust in those models given the nature of Deep Learning paradigms?

To answer this question, we propose defining frameworks based on particular testing approaches that are tailored for different DL paradigms and limitations of testing techniques to cover a large spectrum of use cases. We refine the higher-level questions into the following sub Research Questions (RQs):

RQ*a*: *How can we reliably detect faults in Deep Reinforcement Learning code?* To address this question, Chapter 4 presents a framework for Mutation Testing (MT) extended to Deep Reinforcement Learning (DRL) called *RLMutation*. This question aimed to improve code testing of DRL models by extending standard testing methods widely used in traditional ST, MT, while accounting for the new paradigm brought by DRL in the testing formulation.

RQ*b*: *How to account for stochasticity of Mutation Testing in Supervised Learning?* To address this question, Chapter 5 describes a framework to revisit MT as defined in previous works called *Probabilistic Mutation Testing (PMT)*. This question aimed to better account for the stochastic effect of DL on the existing MT approaches by using a probabilistic approach based on Bayesian probability.

RQ*c*: *How to transfer test sets across Deep Learning models to reduce testing cost while preserving effectiveness?* To address this question, Chapter 6 characterizes a framework called *Generated Inputs Sets Transfer (GIST)* for the effective transfer of test sets. This question aimed to investigate to what extent existing test sets obtained through testing techniques could be reused on a new model under test by exploiting similarities between DL models to reduce the cost of applying testing techniques.

RQ*d*: *How to assess individual tasks within benchmarks of Large Language Models?* To address this question, Chapter 7 defines a framework for determining the difficulty of programming tasks within benchmarks of Large Language Models named *DiffEval*.

The goal of this question is to highlight the limitation of global metrics for the evaluation of Deep Learning models, such as Accuracy, by evaluating the behaviours of the models on individual tasks, with the particular use case of quantifying difficulty. Difficult tasks translate to impactful faults for Large Language Models, which could foster their improvement.

1.2 Contributions

This thesis contributes to ST and DL by providing practical frameworks to enhance the trustworthiness and effectiveness of testing techniques applied to DL models. In particular, the questions addressed in this work tackle a range of limitations of testing techniques and DL paradigms. The key contributions, i.e., the four different frameworks as well as the analysis supplementing them, include:

Contribution 1: *RLMutation*, a framework for Mutation Testing to detect faults in Deep Reinforcement Learning

This contribution extends MT to DRL by redefining the different concepts related to MT, such as killing criteria, test cases and mutations within the frame of DRL. The study answers three questions:

RQa_1: *What are the limitations of existing mutation detection definitions when applied to DRL?* This question highlights the limitations of previously proposed mutation criteria and how to tackle them by defining new criteria. We particularly compare the number of mutations detected by each criterion. Previous criteria show a lower number of mutations detected compared to our proposed criteria.

RQa_2: *How are different agents and environments affected by the different mutations?* In this question, we aim to explore the mutations that can be detected among our sets of defined mutations leveraging the test environments we automatically generated. In particular, we show that patterns in which test environments detect specific mutations could be prolonged into a fault-localization technique.

RQa_3: *Do the Higher Order Mutations generated from our First Order Mutations possess the subsuming property similar to traditional software engineering?* Finally, in this question, we explore whether defined criteria could detect more complex mutations called Higher Order mutations. We generate such complex mutations by coupling defined simple mutations (First Order Mutations) and calculating the number of Higher-order

mutations that can be detected. Our proposed criteria are able to detect Higher Order mutations.

Contribution 2: *PMT*, a framework for accounting for stochasticity of Mutation Testing approaches in Supervised Learning

This contribution extends MT as described in previous approaches to better account for the potential impact of stochasticity of DL by using a Bayesian probabilist framework. The framework uses multiple instances of the same models combined with Bayesian probability and bagging approaches to model the distribution representing the probability of detecting a specific mutation. From this distribution, statistics can be used to inform the detection decision, and we propose a criterion based on those statistics in order to provide a simple but more reliable way than traditional mutation testing to detect mutations. The study answers two questions:

RQb_1 *What are the benefits of the application of PMT compared to simple MT?* This question aims to highlight the difference between simple MT and our proposed probabilistic version by exploring the difference between the two methods in terms of stability, consistency and fine-graininess of the analysis of the mutations, where our probabilistic approach should prove to improve on those aspects.

RQb_2 *What is the trade-off between the approximation error and the cost (sampling size and number of bootstrapped replications) of our method?* In this question, we quantify the error the different parts of our probabilistic approach can lead to, particularly concerning the trade-off with the number of DL model instances needed in the computation. A larger number of instances should lower the error at the expense of more model instances needed, thus resulting in a higher computation time.

Contribution 3: *GIST*, a framework for transferability of generated test inputs for decreasing the cost of testing

This contribution defines test sets' transferability as a new avenue to mitigate generating testing techniques' cost in DL while keeping their effectiveness in terms of the property the test sets aim to cover (e.g., faults detected) by leveraging similarities between available DL models. The process leverages available DL models and their test sets to validate chosen similarities in a "leave one out" approach, simulating one of the DL models being the model

under test for which we want to transfer test sets. Once validated, we can use the empirically validated similarities to transfer effectively on any new model under test without reusing testing techniques on this model, reducing cost.

RQc_1 *How do similarities and properties relate to each other in terms of DNN model types?* This question investigates the relationship between DL models' similarities and the properties, such as faults detected by the test sets. This investigation is supported by the hypothesis that similar models should have test sets covering similar properties. Thus, we should observe higher property coverage on similar models, which could vary depending on the actual similarity metric used.

RQc_2 *Can the introduced similarities be used as a proxy for the defined properties?* This question aims to validate that the similarities investigated previously can act as a proxy for the properties of the test sets we aim to transfer. We define a proxy as an alternative metric which is linked to the property we measure for the transfer. Indeed, as the goal is to mitigate testing techniques application, one would not have access in practice to the test set properties during the transfer. Instead, we would rely on the validated proxy and its similarities to operationalize the transfer. If similarities can be used in such a way, there should be a positive monotonic relation between (some of) them and the property covered by the test sets.

RQc_3 *Can similarities be used to operationalize test transfer?* This question aims to show that, leveraging the defined similarities, we can not only transfer the test sets but also do so in a way that the chosen transferred test sets cover as much of the desired properties as possible compared to any other available test sets. If the transfer can be operationalized, the chosen transferred test sets based on the validated similarities should result in a higher property coverage than any other test sets.

RQc_4 *Is transfer with GIST more effective than generating test cases from scratch from a trade-off property covered/execution time point of view?* This question aims to analyze the trade-off between the covered property of the transferred test sets and the actual computation time it takes to apply the procedure. This measurement will be based on effectiveness ratio metrics between two quantities: the covered property of the transferred test sets and the time ratio between applying the transfer procedure and using the testing technique. This allows us to analyze the relevance of applying transferability depending on diverse scenarios. For the transfer to be useful, we should observe an effectiveness ratio above 1, illustrating applying the transfer is more effective than

reusing the testing techniques.

Contribution 4: *DiffEval*, a framework for individual assessment of benchmarks' tasks for Large Language Models

This contribution analyzes how evaluation based on benchmarks is done, with the particular use case of assessing benchmarks' task difficulty in Large Language Models (LLMs) for code. In particular, we show the limitations of global evaluation metrics and propose an analysis based on the individual tasks within the benchmark instead. This allows for a more comprehensive assessment of models. Moreover, this can further improve models and benchmarks by facilitating the generation of new targeted tasks based on this analysis. The process relies on using two prompt transformations, controlling the rephrasing and the level of information in the prompt, to generate diverse ways of prompting models for the same task. Those transformed prompts probe diverse Large Language Models, allowing us to calculate a score per code sample using code-based synthetic and functional similarity. In the end, a difficulty score for a given task is computed based on those code sample scores, which represent the task's difficulty.

RQd_1 *Does the level of information in a prompt correlate with the score of the generated code samples?* In this question, we aim to validate the level of information transformation, rephrasing having already been shown to impact outcomes in the literature, by calculating the correlation between the level of information and the score obtained for the generated samples. If the level of information impacts the score obtained, we should obtain a positive monotonic correlation.

RQd_2 *Can DiffEval identify hard tasks in a benchmark?* This question aims to compare global evaluation metrics and the proposed approach when it comes to judging whether an individual task can be considered hard. This should translate to tasks labelled as hard by our approach, resulting in low accuracy when using global metrics. However, tasks labelled hard according to global evaluation would not necessarily translate to a high difficulty score using our approach.

RQd_3 *How difficult are tasks contained in the studied benchmarks for LLMs?* This question explores the different tasks within the analyzed benchmarks using our proposed difficulty score. We do so both in terms of the task distribution in terms of difficulty score as well as how they are distributed across different topics. Said topics will be calculated using topic-modelling approaches to cluster tasks (hard and not hard) together to potentially identify topics that are inherently difficult for LLMs.

RQd_4 *Can DiffEval support the generation of targeted new difficult tasks?* This question determines if generating further new hard tasks leveraging previously identified topics and hard tasks is possible. In particular, the approach uses methods referenced in the literature to generate new tasks but accounts for the fact that we aim for specific hard tasks. If the approach is effective, the new tasks generated should be hard according to our approach, that is, having a high difficulty score.

The above research questions developed in this thesis led to the publication/submission of the following research papers:

- Tambon, F., Khomh, F., & Antoniol, G. (2023). A probabilistic framework for mutation testing in deep neural networks. Information and Software Technology, 155, 107129. [19]
- Tambon, F., Majdinasab, V., Nikanjam, A., Khomh, F., & Antoniol, G. (2023, April). Mutation testing of deep reinforcement learning based on real faults. In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 188-198). IEEE. [20]
- Tambon, F., Khomh, F., & Antoniol, G. (2023). GIST: Generated Inputs Sets Transferability in Deep Learning. ACM Transactions on Software Engineering and Methodology. [21]
- Tambon F., Nikanjam A., Khomh F., and Antoniol G., "Assessing programming task difficulty for efficient evaluation of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21227 [22].

The complete list of papers produced during the Ph.D's degree studies can be found in Annex A.

1.3 Organization of the Thesis

This thesis is organized into eight chapters. Chapter 2 introduces key concepts related to the topics tackled in this thesis, namely Deep Neural Networks (DNNs), the cornerstone of DL, the different paradigms of DL, as well as how DNNs are trained. Moreover, it describes what testing is, how it is applied in DL, and what the limitations of testing DNNs are, which this thesis aims to address. Chapter 3 gives an overview of the literature and related works. We

detail aspects that Chapter 2 defined, such as testing methods for DNNs and faults encountered in DNNs. Based on this background, Chapter 4 to Chapter 7 describe the contributions of this thesis, i.e., the four frameworks, describing which DL paradigms they cover, what testing limitations they tackle and how they apply. Chapter 4 presents the first contribution, *RLMutation*, the framework which extends MT for DRL code. This is followed by Chapter 5, which extends MT by accounting for MT limitations in former approaches by defining a framework based on a probabilistic method named *PMT* (Probabilistic Mutation Testing). Those two chapters deal with faults that are within the code, architecture or data of the DL models. Chapter 6 and Chapter 7 instead deal with faults that are within the DL models and linked to the stochastic training process. Chapter 6 introduces the *GIST* (Generated Inputs Sets Transferability) framework, which aims to reduce the cost of testing techniques by leveraging similarity between DNNs for transferring test sets. Chapter 7 deals with the last contribution, the framework *DiffEval*, which shows the limitation of global evaluation and proposes instead individual assessments of the tasks for the use case of programming tasks difficulty in benchmarks of LLMs. Finally, Chapter 8 concludes this thesis by summarizing the contributions and highlighting limitations of the proposed contributions before underlining possible future works to enhance further the trustworthiness and effectiveness of testing techniques in DL.

CHAPTER 2 BACKGROUND KNOWLEDGE

This Chapter presents the general knowledge necessary to understand the contributions and the rest of the thesis. First, we present Deep Neural Networks (DNNs), the cornerstone of DL, through the different architectures of the models used in this thesis. Then, we present the main DL paradigms used in this thesis as well as the general idea of how DL models are trained on data, making them differ from traditional programs. Finally, we delve into what it means to test a program, starting with the general definition used in ST, before seeing how DL affected those definitions. We also define the main avenues for testing DNNs and how such testing techniques are limited due to the nature of the DL paradigms. In the end, we give a table to summarize how each of our frameworks presented in this work contributes to tackling those limitations by defining for each of them which faults, paradigms and testing issues they deal with.

2.1 Deep Neural Networks

The increase in computation power saw the improvement of simpler Machine Learning (ML) models to DNNs. In the following, we describe the main DNNs architectures used in this thesis.

2.1.1 Feedforward Deep Neural Networks

Feedforward DNNs, sometimes called Multilayer perceptron, were the first type of DNNs introduced. Feedforward DNNs aim to learn from available data D to approximate a function f [23], which should represent the underlying (yet unknown) phenomenon of D.

Feedforward DNNs are composed of a composition of functions $f^{(j)}$ called *layers*. In the example given in Figure 2.1, the DNN is composed of two layers plus the input layer, thus two functions $f^{(j)}$. The more layers, the "deeper" the DNNs.

In the most simple form, each function $f^{(j)}$ is represented as a linear combination of a weights $\boldsymbol{\theta}_j = \theta_{j,1}, ..., \theta_{j,n}$ and inputs $\boldsymbol{x} = x_1, ..., x_n$, eventually with a constant term b_j named a *bias*, such as $f^{(j)}(\boldsymbol{x}, \boldsymbol{\theta}, b_j) = \boldsymbol{x}^T\boldsymbol{\theta}_j + b_j$. For instance, for the first layer, this would translate to $f^{(1)}(\boldsymbol{i}, \boldsymbol{\theta}_1, b_1) = \boldsymbol{i}^T\boldsymbol{\theta}_1 + b_1$. For the subsequent layers, the inputs $\boldsymbol{x}$ are the inputs of previous layers; thus for the second layer $f^{(2)}(f^{(1)}, \boldsymbol{\theta}_2, b_2) = (\boldsymbol{i}^T\boldsymbol{\theta}_1 + b_1)\boldsymbol{\theta}_2 + b_2$. Hence, these DNNs are called “feedforward” as the information flows from the input layer to the output layer in one direction.

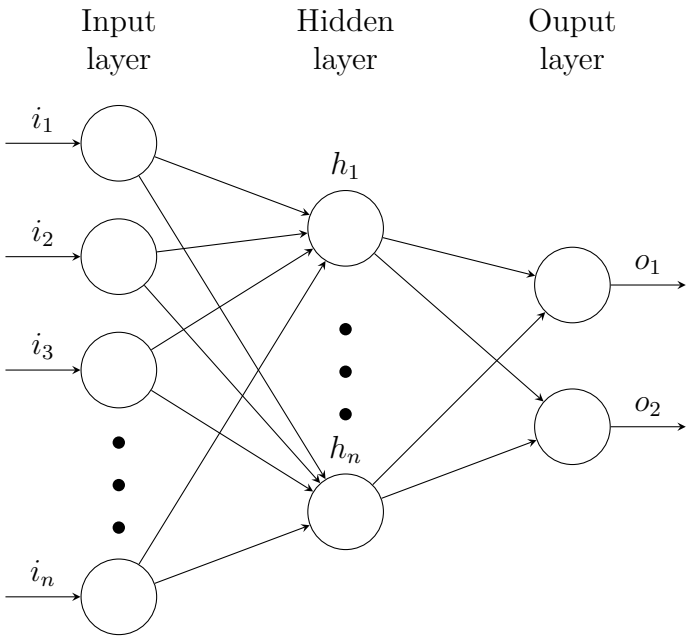


Figure 2.1 Example of Neural Network

Those linear functions are, however, not enough. For instance, following this definition, DNN would be linear as a composition of linear functions. Nonetheless, this would drastically limit their capacity, as they would be, for instance, incapable of solving the XOR problem as it is non-linear. To introduce non-linearity, an activation function g is generally used after each layer. Thus, in practice for the first layer, we would instead have $f^{(1)}(\boldsymbol{i}, \boldsymbol{\theta}_1, b_1) = g(\boldsymbol{i}^T\boldsymbol{\theta}_1 + b_1)$. Several activations function g can be used, yet the most common ones are the REctified Linear Unit (ReLU) [24] which is defined as $g(\boldsymbol{z}) = max\{0, z\}$ or the Softmax/Sigmoid which are defined for each element z_k of $\boldsymbol{z}$ as $g(z_k) = \frac{e^{z_k}}{\sum e^{z_p}}$ and $g(z_k) = \frac{1}{1+e^{-z_k}}$.

2.1.2 Convolutional Deep Neural Networks

Convolutional DNNs [25] are an improved version of Feedforward DNNs that are adapted to deal with grid-like topology, most notoriously 2D data such as images [23]. Convolutional DNNs' main feature is the use of convolution operations in convolutional layers. We give an example of convolution in Figure 2.2. Given an input (for instance, an image), a kernel is applied by sliding it over the image and computing the sum of the element-wise product between the kernel and a slice of the image corresponding to the current sliding. Parameters such as kernel size or sliding length are defined when creating the DNNs, while the actual weights of the kernel are the θ parameters that will be learned on the data.

The main arguments of convolution are that it helps preserve both the local structure and

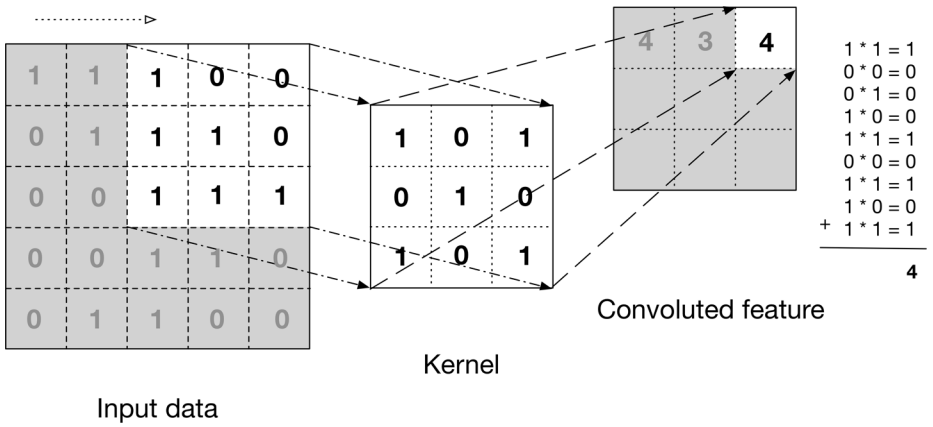


Figure 2.2 Example of convolution [1]

contribute to the equivariance of the translation of the DNN; that is, a translation of an image should change the output in the same way. On top of that, this architecture allows us to reduce the number of weights needed, as instead of having one weight per input, as is the case for Feedforward DNNs, the kernel weights are shared across inputs.

On top of Convolutional layers, Convolutional DNNs make use of Pooling layers. Pooling layers summarize the local output with some statistics. For instance, max pooling [26], which is the most used pooling operation, returns the highest value in a given rectangular neighbourhood. Similarly to the convolution kernel, this neighbourhood has a defined size and sliding window. Pooling operations make the representation approximately invariant to a small translation of the input [23]; if the input is slightly translated, the output should not change. Those properties are particularly interesting when dealing with images.

Based on those two principles, one can similarly create a Convolutional DNN to the Feedforward alternative presented earlier. An example of such DNN is given in Figure 2.3.

Convolutional DNNs combine convolutional layers with pooling layers to transform a given input (in the example, an RGB image size 32 per 32) into a collection of feature maps of smaller sizes. Generally, those feature maps are flattened to fall back into more traditional feedforward layers until the output, particularly when used in supervised learning (see Section 2.2).

2.1.3 Transformer architectures

While previous architecture could tackle sequential data (such as text), they were not specifically designed for such a task. While architecture such as Recurrent DNNs or Long-Short Term Memory DNNs were developed to tackle those data, the arrival of Transformers DNN [2]

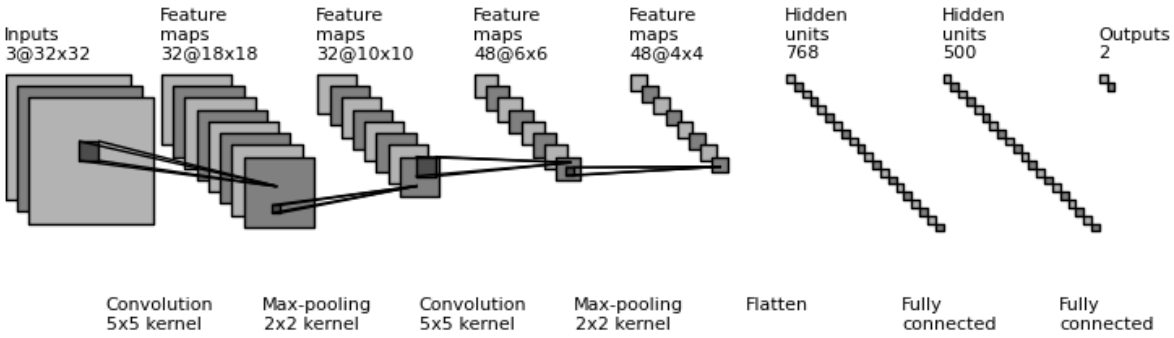


Figure 2.3 Example of Convolutional DNN

enhances the capacity to deal with those data. In the following, we present a high-level explanation of Transformers, so readers understand the general concepts used in this thesis. For more information on the inner workings of the architecture, for example, the encoders and decoders blocks mentioned in the next paragraphs, we invite the readers to confer to the following references [2, 27–29].

The crux of the architecture relies on the *attention* mechanism, which, at a high level, helps the model to look at different parts of the inputs and how they relate together. An attention layer is defined as follows: three matrices are initialized called the Key (Θ^K), Query (Θ^Q) and Value (Θ^V) matrices. We can obtain an input's K, Q and V vectors by multiplying the input vector with each matrix. For instance, in the case of text, we can multiply the vectors representing each word in the text by the matrices Θ^K, Θ^Q and Θ^V. Then, the vectors K and Q are multiplied with a dot product before scaling their results by dividing by a constant (the square root of the K vectors dimension) and using a softmax operation we introduce in Section 2.1.1. The softmax roughly highlights how meaningful a part of the input is given the context. The vectorized softmax is then multiplied by vector V to focus on relevant parts of the input and neglect irrelevant parts. Finally, the output of the attention layer is the sum of the weighted vectors V. The attention mechanism is represented in Figure 2.4, and more details can be found in the original paper by Vaswani et al. [2].

In practice, multiple attention heads are used, allowing the model to focus on different aspects and obtain different representations of the same words through different initialization. The multiple attention heads are simply different attention heads following the mechanism defined previously, for which we concatenate the output.

Building on those attention layers, a transformer DNN can be obtained, for which we give a representation in Figure 2.5.

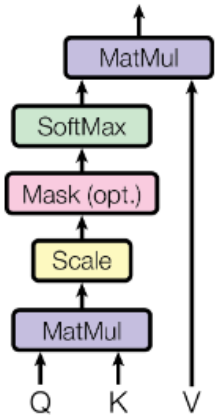


Figure 2.4 Attention mechanism [2]

The general architecture is based on blocks of encoders (left) and/or decoders (right). At a higher level, encoders are tasked with learning a latent representation of the inputs given, while the decoder needs to reconstruct an output based on those encodings. First, an input, for instance a text, is given to the DNN. This text is generally split into parts (called tokens), which are then embedded in the model to be understood using an embedding layer. As Transformers do not have a recurrence mechanism like Recurrent DNN, they also use Positional Encoding to keep track of the place of the word in the text. Those embeddings are combined and fed to the blocks of encoders, where all encoders follow the same architecture: a combination of multiple attention heads as explained previously, followed by a Feedforward DNN as detailed in Section 2.1.1. The blocks of decoders follow a similar architecture, except that each decoder has two multiple attention head blocks. One is used to feed the output embedding, mirroring the input embedding. For instance, if our input is a question, the output would be the answer, and so this output embedding would contain the embedding for each word in the answer. While it is empty initially, it will be updated as the answer is generated. The second attention block is used on top of both the previous decoding block's output and the encoders' block's output. The general process involves encoding the input text into a lower-level representation to use this representation along with current output embedding to generate the next token. This process is repeated multiple times, generally until the end of the answer has been reached (symbolized by a specific token), or the limit of the generation has been exceeded. Transformer DNNs are fairly recent but are of particular importance as they are the main building blocks of Large Language Models (LLMs) such as ChatGPT [5].

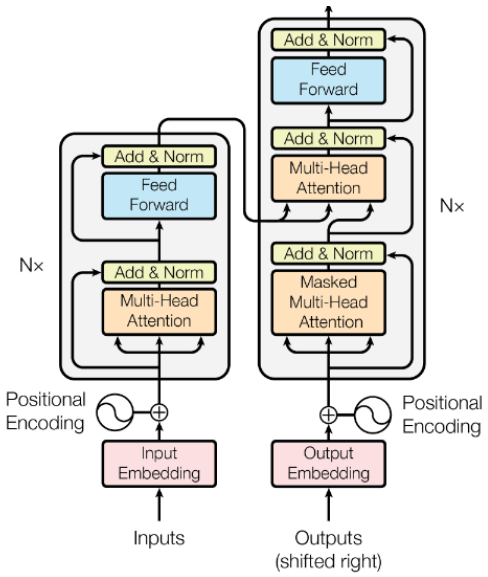


Figure 2.5 Transformer general architecture [2]

2.2 Deep Learning paradigms and Deep Neural Networks training

Having described the different architectures of DNNs used in this thesis, we present the DL paradigms we will use and how DNNs are trained.

2.2.1 Deep Learning Paradigms

Supervised Learning

In supervised DL, we consider our data D to be composed of couple (x, y), where x is an input (features describing an object, an image, a text etc.) and y is a target, which can be either discrete (for instance, labels such as "cat" or "dog") or continuous (for instance, the angle of the wheel in a self-driving car). The goal is to find a function f that best approximates the mapping of the x to their target y. The learning is supervised as we have some desired targets that are known in advance and for which we would want the function f to map correctly. An example of such a paradigm could be a Convolutional DNN trained to classify images based on known labels such "cat". In that case, the output of the DNN becomes a score for each label representing the "likelihood" that the image belongs to this label according to the DNN. Here, Softmax activation (see Section 2.1.1) is helpful, as it can map any vector to a vector for which the dimensions are in $[0, 1]$.

Unsupervised Learning

Unlike Supervised Learning, Unsupervised Learning aims to find the function f without supervision, i.e., without any y. Thus, where supervised learning would try to model $p(y|x)$, that is, the distribution of the label y knowing the data x, unsupervised methods aim to model $p(x)$, which is the data distribution itself. Unsupervised learning is broad and ranges from clustering data based on similarity to generating new data based on the learned distribution. For that last task, multiple have been proposed: Variational Autoencoder DNNs [30], Generative Adversarial DNNs [31] or, more recently, Transformer DNNs that we saw previously.

Reinforcement Learning

Reinforcement Learning behaves somewhat differently from the two other paradigms because the notion of data is different. In Reinforcement Learning (RL), an agent interacts with an environment, for instance, a robot that needs to reach the end of a room while avoiding obstacles. Concretely, at each time-step, t, the agent perceives the state s_t it is in (e.g., position in the room, whether it hit an obstacle, etc.) and takes action a_t (e.g., moving right, left, forward etc.) accordingly. Then, the agent gets a reward r_t depending on the result of the action (e.g., -10 for hitting an obstacle, +1 for not, etc.) before transitioning into the next state s_{t+1}. The process continues until the agent reaches either a *terminal* state or a certain number of *steps* (i.e. actions/states transitions) has been taken, in which case the environment ends. We call *episode* the length of the experiences the agent collects, starting from an (initial) state and ending when the environment finishes. As such, there is no data as we would have in supervised learning; instead, it is an environment from which the agent learns. Said agent is generally controlled by a *policy* DNN, which will return scores for each possible action given the agent's current state. RL aims to learn a policy π which will maximize the expected cumulative reward the agent can get, in our robot example, by reaching the end of the room while avoiding all obstacles. Generally, we want our agent to learn the trade-off between short and long-term rewards. To do so, we use the concept of *discount factor* γ, which is used when computing the cumulative rewards such as $\sum_t^\infty \gamma^t r_{t+1}$.

We give a representation of the three paradigms in Figure 2.6.

2.2.2 Training Deep Neural Networks

The previous sections described the DNNs' architecture, how they work, and the different DL paradigms. However, we did not mention how the different weights (θ or Θ) within the

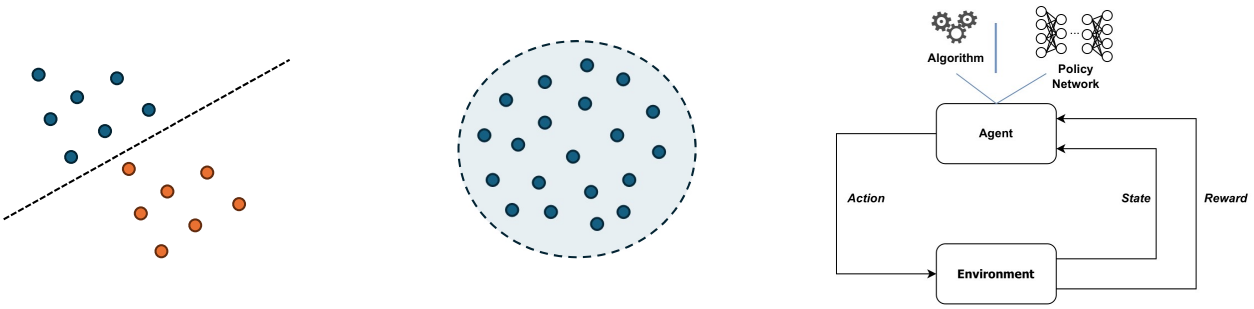


Figure 2.6 DL paradigms: Supervised (Left), Unsupervised (Center), Reinforcement (Right)

DNNs were obtained. All DL paradigms rely on initializing those weights randomly and using data to "learn" those weights. At a high level, all the paradigms we described require some objective that we call a *Loss* function $\mathcal{L}$. For instance, in the case of supervised learning, the loss function would measure the distance between the score obtained for the different labels and the ground-truth label value y. In that case, the *cross-entropy* loss is generally used to define for a given input x as $-\sum ylog(o)$ where y is the target and o the prediction of the DNN using the approximate function f. In practice, the label y is represented as a one-hot encoding vector of size n, the number of labels possible, such as $y_j = 1$ for the label of x and $y_j = 0$ otherwise, and o is represented similarly as a vector of predictions with score ranging from 0 to 1 such as the sum over all dimensions of the vectors sum to 1.

Once $\mathcal{L}$ is computed, the next step is propagating changes. Indeed, the loss informs us how far the computed value through the approximation f is from the actual ground truth. That is, there is a difference δ that we would like the DNN to correct in its approximation. To do so, we need to propagate the changes in the different weights of the models to acknowledge this δ. To do so, the most used algorithm is the *Backpropagation* algorithm [32], which relies on computing the gradient of the loss, helped by the chain rule of calculus [23].

Remember that, from Section 2.1.1, we had a feedforward DNN which we could express for an input $\boldsymbol{i}$ as $f(\boldsymbol{i}) = g_2(g_1(\boldsymbol{i}^T\boldsymbol{\theta}_1 + b_1)^T\boldsymbol{\theta}_2 + b_2)$ where $\boldsymbol{\theta}_j$ are the weights for the j^{th} layer, b_j the bias for the j^{th} layer and g_j the activation function for the j^{th} layer. In that case, using backpropagation, we can update the weights $\boldsymbol{\theta_j}$ and biases b_j following $\boldsymbol{\theta}_{j,new} = \boldsymbol{\theta}_{j,old} - \alpha\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}_j}$ and $b_{j,new} = b_{j,old} - \alpha\frac{\partial\mathcal{L}}{\partial b_j}$, where α is a parameter called the *learning rate*. Suppose we consider g_1 as the ReLU activation function and g_2 as the Sigmoid activation function. The loss is the cross-entropy function we presented earlier in the case of a supervised learning algorithm. In that case, we can derive the $\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}_j}$ needed to update the parameters in each layer. For example, we show how to update compute $\theta_{2,1}$, the second layer's first weight, linking the hidden layer's

first unit to the output layer in Figure 2.1. To simplify notation, we consider h_j to be the input of the hidden layer neurons, z_j to be the pre-activation (before Sigmoid) output of the hidden layer neurons and o_j their output post-activation. We derive $\frac{\partial \mathcal{L}}{\partial \theta_{2,1}}$ as follows:

$$\frac{\partial \mathcal{L}}{\partial \theta_{2,1}} = \frac{\partial \mathcal{L}}{\partial o_1} \times \frac{\partial o_1}{\partial z_1} \times \frac{\partial z_1}{\partial \theta_{2,1}} + \frac{\partial \mathcal{L}}{\partial o_2} \times \frac{\partial o_2}{\partial z_1} \times \frac{\partial z_1}{\partial \theta_{2,1}} \tag{2.1}$$

$$\frac{\partial \mathcal{L}}{\partial o_j} = \frac{y_j}{o_j} \tag{2.2}$$

$$\frac{\partial o_1}{\partial z_1} = \frac{e_1^z}{e_1^z + e_2^z} + \frac{(e_1^z)^2}{(e_1^z + e_2^z)^2} = o_1 - o_1^2 \tag{2.3}$$

$$\frac{\partial o_2}{\partial z_1} = \frac{-e_1^z e_2^z}{(e_1^z + e_2^z)^2} = -o_1 o_2 \tag{2.4}$$

$$\frac{\partial z_1}{\partial \theta_{2,1}} = h_1 \tag{2.5}$$

Equation 2.1 shows the branching needed to backpropagate and apply the chain rule correctly, as two outputs can affect our update. The rest is derived using the definition of the activation functions and our notations. Putting back everything together leads to the following:

$$\frac{\partial \mathcal{L}}{\partial \theta_{2,1}} = (y_1 - o_1)h_1 \tag{2.6}$$

$$\theta_{(2,1),new} = \theta_{(2,1),old} - \alpha(y_1 - o_1)h_1 \tag{2.7}$$

The same can be done to update the rest of the weights and biases. The overall training procedure consists of updating weights/biases several times. To track the performance and the improvement of the DNN, ensuring that it is learning, it is customary to have some data (called a *test set*) separated from the data that are used to train the model (called a *train set*). Using this test set, we can measure the generalizability of what the DNN is learning. Indeed, relying only on the performance measured on the train data is unreliable as the DNN can start to "memorize" those data and thus have a function f which will work very well on those train data but terribly on other data. For Reinforcement Learning (Section 2.2.1), as the notion of a test set does not exist, we instead rely on the expected cumulative reward obtained during the training progresses. This general scheme is similar for all DNNs and all paradigms, requiring adaptation to the specificity of the current DNN and/or paradigm used or the loss function chosen.

2.3 Testing in Deep Learning

2.3.1 Definition from traditional Software Testing

Testing is a central part of any software engineering practice and is a field of research on its own. Testing is essential not only to improve the quality of a program but also to assess its validity. One of the main goals of testing is to detect *faults*, for instance, a condition wrongly implemented, which is the cause of *errors* in program [33]. If those errors are not corrected, *failures* will occur, defined as an observable behaviour of the program that does not conform to the expected specification. Thus, finding those faults generally implies detecting failures and tracking down their root cause to fix them. This thesis focuses on the first part, i.e., detecting the failures through testing methods.

Testing relies on *test cases*, that is in the most basic form a couple of $(input, expected\_output)$, where the expected output is what we aim our program to achieve given the input. An ensemble of *test cases* is generally referred to as a *test set* or as a *benchmark* if the test cases should be used as standard measurements for multiple programs. This notion is similar to the (data, label) parts used in supervised learning. Those test cases are used to find potential failures that signal probable faults. The execution of those test cases can be done at multiple levels of a program: from *unit testing* that deals with verifying the code of the program itself to system testing, which aims to test higher levels of the program such as security, robustness, stability, etc.

There exist multiple testing techniques, which are generally sorted into three broad families based on what information is required to apply the test [34]: white-box testing, black-box testing and grey-box testing. Black-box testing is the simplest yet most widespread form of testing, dealing essentially with the functionality of a program. As such, it does not rely on the internal structure of a program. One popular black-box approach is called *fuzzing* [10], a testing method that aims to detect faults by providing the programs with a multitude of random inputs. On the other hand of the spectrum, white-box testing assumes access to the internal structure of the program and so is more potent than black-box testing. One example of such a technique is branch coverage testing, which verifies how many branches of a code a specific test suite triggers. A branch is a decision point in the source code, for instance, when an "if..else.." structure is used. Logically, the higher the proportion of branches covered, the more certain we are that we tested our code adequately, so the less likely we can encounter a fault. Between those two concepts lies grey-box testing at the intersection. In that case, *some* knowledge of the program's inner workings is assumed, but it is not complete. For instance, during integration testing, when two parts of a program are tested jointly and, each

of those parts can only access the other through the exposed interfaces.

Testing, however, is not without limitations. As testing relies mainly on test cases, the quality of such test cases is essential for adequately testing a program. If one corner case that needs to be tested is missed, a fault can be neglected, and the program can still be deemed correct. Thus, the main limitation of testing can be summarized as "Program testing can be used to show the presence of bugs, but never to show their absence" [35].

2.3.2 Impact of the Deep Learning paradigms on Testing definitions

Given the previous definition, we now set to specify in what measure the definitions apply to DL. In DL, faults materialize in two main ways. First, at the specification-level, where we consider that the specification includes both the DNN code and the data it is learning from. In the case of code fault, this is similar to traditional programs, as DNNs remain fundamentally programs, and so faults can occur in code implementing DNNs (wrong implementation of an algorithm, bad parametrization, etc.). Regarding data-related faults, those are specific to DNNs and can happen when data are poorly handled (corrupted/noisy data, mislabeled data, missing data etc.). Note that, in the case of faults within data, we only consider a fault if the lousy handling of the data was not intentional. As such, methods such as data augmentation, which add modified/noisy data on purpose to make DNNs more robust, are not considered to be a specification fault. The second type of fault is at the model-level and is inherent to the DL paradigm as DNNs' internal logic is not coded but "learned" on the data. In that last case, multiple aspects of the DL paradigms can be a fault, such as the effect of the stochastic learning, under-specification and inductive bias of the DNNs, and training process based on minimizing an empirical loss on a limited number of data, etc. For those model-level faults, faults can impact a specific DNN instance or affect a whole class of DNNs. For instance, in the first case, it is possible that different DNNs trained on the same data, architecture, and code but with a different seed will not yield the same faults because of stochastic learning. In the second case, this would affect all DNNs in a class of models. For instance, LLMs generally involve some forms of alignment training following a pre-training, which can lead to faults if the pre-training is not done adequately [36]. Faults at the specification-level are generally more concrete as they are linked to direct code errors or data errors. On the contrary, faults at the model-level are more abstract because of their origin within the DL process itself. In any case, those faults generally lead to failures in DNNs.

Failures in DNNs manifest similarly in two ways. First, failure can be expressed through the DNNs crashing or hanging, similar to traditional programs. This is generally linked to

a fault in the code of the DNNs, and this type of failure is easily identifiable. The second failure type, more widespread and insidious, generally occurs through DNN outputting a wrong result (i.e., different from the expected one). Such failures can take specific forms depending on the DL sub-paradigm employed: the failures for Convolutional DNNs trained on classification tasks would result in mispredictions (e.g., a dog image being predicted as a cat) while the failures for LLMs could manifest through hallucinations (i.e., a "generated content that is nonsensical or unfaithful to the provided source content") [36]. In that last case, the faults can originate either from a specification-level or model-level issue. We give a summary of the failures and faults in DL in Figure 2.7.

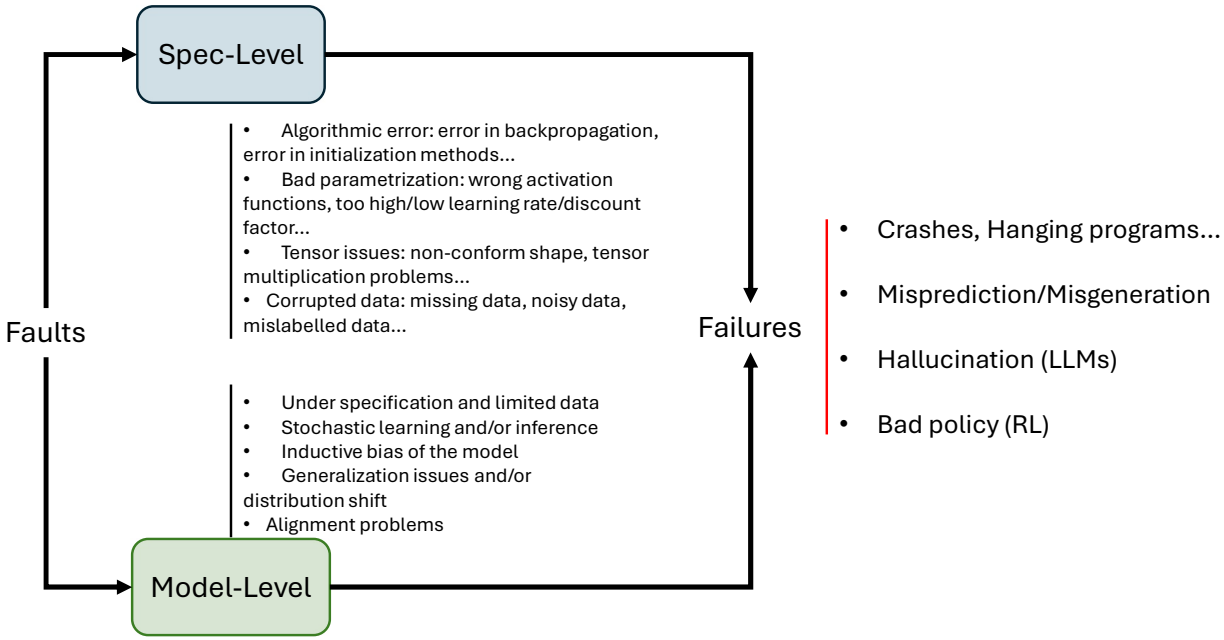


Figure 2.7 Failures and Faults in Deep Learning

As examples of faults in DNNs we studied, we mention our work [37] related to *silent bugs* in DL frameworks, which highlight the problematic for specification-level (code) faults, where we studied how silent bugs are affecting DL frameworks and users' code. In our context, silent bugs are bugs that do not result in apparent failures such as crashing or hanging. While such bugs existed in traditional programs and were particularly pernicious, they are even more so in DL because of the nature of failures. For instance, one type of bug that we identified affected the DNN model's saving by not correctly registering the weights. Consequently, users would train a DNN and test it to ensure everything works fine and saves weight. However, when they load the weights again for inference on new data, the performance of the models

drastically decreases. Because of the potentially diverse cause of failures for DL we described earlier, it might not be evident to the user that the faults were 1) in the code and not either in data used for the inference or in the model, as the failure could have been caused by badly resetting the DNN model or the inference mode, 2) that the fault is indeed within the framework the user is using, and not the user's code. The fact that such bugs made it to production shows that no test managed to catch it.

2.3.3 Testing techniques for Deep Neural Networks

Based on existing ST approaches, several testing methods were developed to evaluate and test DNNs. We present a quick overview of the main testing methods.

Testing metrics

The most straightforward way of testing a DNN is to rely on metrics computed using the output of the DNN on available data to evaluate said DNN. As explained in the previous section, a test set explicitly separated from the data used for training the DNNs or a benchmark is generally used. Regarding the metrics, the most popular ones remain the Accuracy (number of inputs correctly predicted over the total number of inputs) or the F1-score (harmonic mean between precision and recall) for classification-based problems or Mean Square Error for regression-based problems (the sum of the squared difference between prediction and expected outcome) [8]. Metrics can also be based on the type of data used: For instance, for image, Intersection Over Union (IOU) can be used, whereas for text, metrics such as BLEU (Bilingual Evaluation Understudy) [38] are used instead. We also mention CodeBLEU [39], which we will use in this thesis in Chapter 7. CodeBLEU is an improvement of the BLEU metric specifically designed for code. Whereas BLEU focuses on N-gram match only, CodeBLEU adds on top of some code-oriented metrics such as Abstract Syntax Tree (AST) (i.e. a tree representation of a code fragment) match or Data-Flow (i.e., a tree representing the variable interaction in terms of values). The different metrics are then aggregated through a weighted average, with the code metrics being given higher weights. We give an example of an AST and Data-Flow graph in Figure 2.8.

Differential Testing

Differential Testing is a broad area that uses multiple objects to evaluate a given model. We highlight only two main groups through N-versioning and metamorphic testing. In N-versioning, a divergence in output on the same input between the programs indicates a fault.

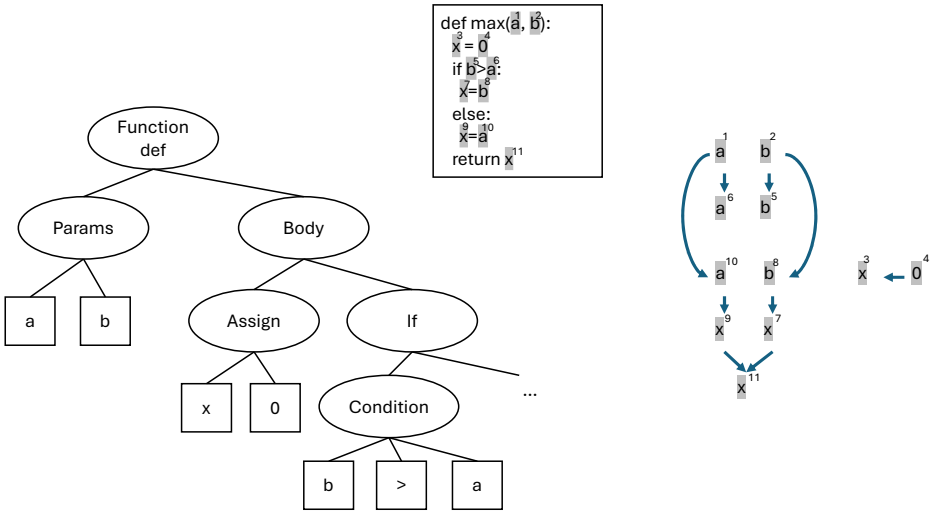


Figure 2.8 For a simple code fragment of a function computing the max between two input values, we can extract the AST Graph (Left) Data-Flow Graph (Right) [3]

In the case of DNNs, N-versioning testing translates to using multiple DNNs to verify their respective output against the same input. This is notably used to create partial-oracle and thus tackle the "oracle problem", which DL models are affected by (see Section 2.3.4). In another direction, metamorphic testing [40] also aims to tackle the "oracle problem" by creating a pseudo-oracle. Yet, instead of focusing on multiple DNN models, metamorphic testing leverages the expected invariance of the task for which the DNN models are trained. For instance, rotating slightly an image of a dog should still be predicted as being a dog by any DNN, as the slight rotation does not fundamentally change the images. By doing so, there is no need to have access to any ground truth for the input, as the ground truth comes from the fact that both inputs (e.g. rotated input and non-rotated) should yield the same prediction through the DNN. Any disagreement highlights a potential fault.

Mutation Testing

Mutation Testing (MT) [9] is an established ST method and a de-facto standard for testing [41]. The goal of MT is not to test a program but rather to test the capacity of the test set

itself to detect faults. It is often used in a regression testing approach, that is, when the code in a program is evolving or when designing a new test. MT is based on two hypotheses: the competent programmer and the coupling effect. The competent programmer hypothesis supposes that developers wrote near correct codes, i.e., the code is not flawless but does not stray far from the target specification. The coupling effect hypothesis supposes that test data that can detect simple errors should be able to detect many complex errors, defined as simple errors coupled together. In practice, MT consists of generating *mutants*, a copy of a correct program where a fault was seeded using a *mutation* operator. Following the hypothesis stated earlier, this fault should be linked to the actual fault a programmer would make. As such, mutations are generally inspired by real faults found in programs. Then, for the given test set, we want to test the mutant programs obtained. Suppose the test set reveals the fault, that is, some inputs lead to a different output than the expected outcome through the mutant. In that case, we say that the mutant is considered to be *killed* (or that the mutation is considered *detected*). On the contrary, if no fault is revealed, then the mutant is said to *survive*. Following the coupling effect hypothesis, mutations can even be coupled together, leading to what is called Higher Order mutations (HOM) [42], as opposed to simple mutations named First Order mutations (FOM). HOM types are defined based on the sets of test cases used to detect the HOMs and the FOMs composing said HOMs. We give Figure 2.9 to illustrate this concept.

A HOM is *Coupled* if the set of test cases detecting the HOM T_H is not empty, and if the intersection between T_H and the union of sets of test cases killing its constituent FOM, $\cup T_i$, is not empty. A HOM is *Subsuming* if the size of T_H is smaller than the size of the $\cup T_i$. The *Subsuming* notion can be refined in *Strongly Subsuming* if T_H is a subset of the intersection of T_i, i.e., $T_H \subseteq \cap T_i$. From the definition, we see that Subsuming HOMs, and particularly Strongly Subsuming, are of particular interest as they lead to mutations that turn out to be more complex than their constituents FOMs, as they are harder to detect. The overarching goal of MT is to verify that we have a test set capable of killing all mutants we can come up with. Some mutants surviving is a sign that the test set might not be good enough and should be improved.

Coverage Testing

With the branch coverage criterion (Section 2.3.1), we saw that some coverage criteria exist in ST. Such criterion could similarly be used to test DNNs. However, in DNNs, code is not the main relevant part to test for when it comes to DNNs' inner logic, as said logic is based on how the DNNs "learn" from the data. As such, testing, for example, for branch or code

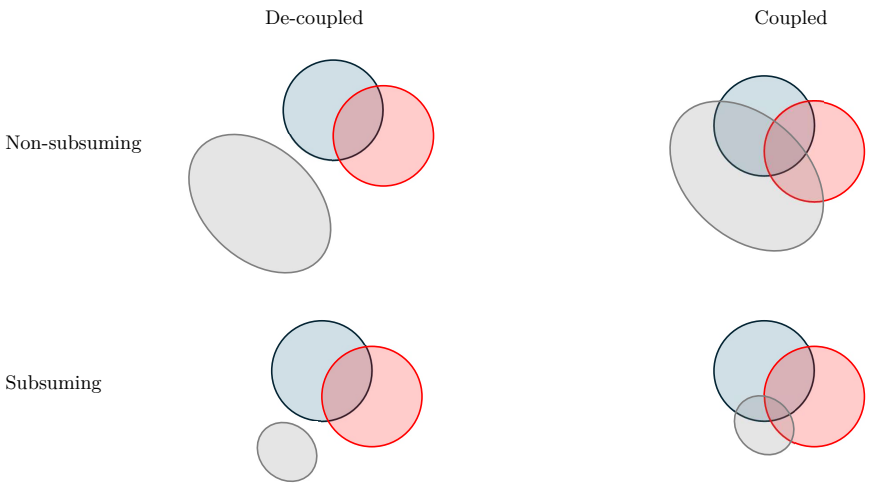


Figure 2.9 HOM types based on the set of test cases that can detect the HOM (in grey), and the sets that can detect the FOMs composing the HOM (in red and blue).

coverage, is rarely meaningful, and it is possible to reach 100% code coverage while the DNN models contain obvious faults [11]. Nonetheless, inspired by this approach, we could assess to which point the inner logic of the DNNs is exercised by a test set by checking how the *neurons* behave. We detail later in the Literature Review (Section 3) the different possible to assess this property, but as an example of coverage criterion for DNNs, we briefly explain Neuron Coverage (NC) [11], the first and most simple coverage criterion. The idea of NC is simply as follows: the more neurons are triggered by a given test set, the more the inner logic of the DNNs is exercised. In that case, calculating the coverage boils down to computing the proportion of neurons with an activation value above a certain threshold when given our test set as input.

Automated Test Generation

Lastly, enhancing existing test sets by generating new ones is possible. Those methods are generally composed of a generator and an objective. The generator is any process that allows the generation of new test sets so that the generated test inputs are still valid from the perspective of testing the DNNs. This can be through random noise (fuzzing just as in traditional ST), using semantic preserving transformations, using adversarial perturbations (with the notion of adversarial attacks [43]) or even using Generative AI through, for instance, Generative Adversarial Networks (GAN) [31]. While an objective is not needed, it is generally included to avoid generating too many test inputs and guiding the generation process. The objective can be any property of interest one would like the generated test inputs to have,

such as the coverage criteria presented before.

2.3.4 Issues when testing Deep Neural Networks

DL and DNNs changed the way ST had to be thought of. The crux of the issue lies in the difference between DNNs and more traditional programs. Whereas, for traditional programs, the inner logic is coded by developers based on a series of requirements, highlighting the role of the code as the specification of the program, the inner logic for DNNs is "learned" using available data and training procedure as we explained in Section 2.2. In that case, the code is only a single part of the specification. The specification is now also contained in the data. More than the code and the data, the complete learning process can also be a source of faults. We described in Section 2.3.2 how this could impact the notions of faults and failures. Testing those specifications then becomes quite a challenge, as it is much more complex than verifying code-based specifications only. Because of this paradigm change, multiple aspects of the testing techniques we mentioned previously can be affected:

The Oracle problem

While some traditional programs could present those issues, the oracle problem is prevalent in DL. Weyuker [44] described a category of program, the so-called "non-testable programs", for which an oracle (the expected outcome of an input through a program) is not existent or is hard to verify. DL programs are part of this category, as the tasks they are trained for are so complex that verifying the outcomes is not doable in a reasonable time and only through human means.

Numerical instability

Numerical instability is not only a DL issue, as it can be found in scientific programs, but DNNs are nonetheless prone to numerical instability. Due to computers' architecture, small perturbations can have a drastic impact on a computation's outcome. Moreover, the precision of variables in memory or parallel computation, most notably with a Graphics Processing Unit (GPU), which DNNs increasingly require, can also lead to such problems.

Stochasticity issues

Because the training process (and sometimes because of the inference process too) is inherently non-deterministic, testing DNNs can become challenging. Indeed, as DNNs' weights are initialized randomly, the training process might converge in different directions if the same

DNNs are trained several times, even though the DNNs' architecture, the data and process are the same every time. This leads to a situation where two DNNs have different predictions on the same input.

Cost of testing

Testing programs can become quite expensive depending on the program's size and how much testing is needed; to that effect, DNNs can be very expensive to test. Obtaining a test set can be quite a manual effort, as it generally requires a human hand to correctly parse and label the data, checking for potential noise. We saw in the previous section that one could generate test inputs; however, this can also lead to issues. Generation can be costly in terms of time and computing resources as guiding objectives require probing DNNs and so are directly affected by the scale of DNNs. Moreover, the generation process should be properly defined and controlled, as it can lead to inputs that are more akin to noise than proper test inputs, rendering the usefulness of the test inputs limited.

Global evaluation over test sets limitations

As mentioned in previous sections, metrics such as Accuracy in conjunction with test sets are widely used in DL as they give a simple summary view of the performance of a model. While useful, such global evaluation might, however, disregard locally interesting information: for instance, a DNN might reach 90% accuracy on a test set but, as argued by Riccio et al. [13], if the 10% left are the most damaging failures, then the DNN is far from adequate for the task. Similarly, for the inputs where the DNNs are correct, slight perturbations might change the predictions, so the global performance is not representative of the actual performance of the DNNs. As such, metrics used in global evaluations of DNNs are limited.

2.4 Overview of Faults, Testing Issues and Paradigms tackled in this Thesis

Given the previous discussions on the types of faults, testing issues and paradigms in DL, we give Table 2.1 as an overview of what each following chapter will tackle as a reference.

Table 2.1 Overview of the faults, testing issues and paradigms tackled by each chapter within the thesis.

	Chapter 4	**Chapter 5**	**Chapter 6**	**Chapter 7**
Faults	Specification Level	Specification Level	Model Level	Model Level
Issues	Stochasticity	Stochasticity	Cost of Testing	Global evaluation
Paradigms	Deep Reinforcement Learning	Supervised Learning	Supervised Learning	Large Language Models
Proposed Framework	*RLMutation*	*PMT*	*GIST*	*DiffEval*

CHAPTER 3 LITERATURE REVIEW

This Chapter gives an overview of works related to the thesis topic. First, we present different ML and DNN models, focusing on the image and text modality that we used in this thesis. Then, we present works related to faults in DNNs, both at the specification-level and model-level. We then expand on testing techniques that were proposed in the literature. Finally, we highlight works that studied the limitations of DNN testing techniques.

3.1 Machine Learning and Deep Learning models

ML is a subfield of AI that deals with algorithms "learning" from data. It originates from statistical and pattern recognition fields. ML encompasses several families of models [45] such as the Perceptron Network [46], from which the Feedforward architecture presented in Chapter 2 is based on, Random Forest [47] which uses an ensemble of decision tree, Support Vector Machine [48] which are supervised max-margin models or k-Nearest Neighbors [49] which relies on distance to classify instances. ML models, however, were limited to smaller datasets and were not adapted when dealing with certain types of data, such as images. The development of new architecture, such as the Convolutional Network [25], which could handle image data, as well as the progress made in computation power through the development of GPU, allowed researchers to focus on deeper neural networks. This marked the beginning of the DL era. DL has been applied to several domains [50, 51], and we only give a brief overview of image and text-based DNNs as they are the ones we are dealing with in this thesis.

Deeper models for image classification were developed, such as AlexNet [52], VGG [53] or ResNet [54]. In parallel, image datasets of increased complexity were produced to support the evaluation of those models from MNIST [55], CIFAR10 [56] to ImageNet [57]. DNNs were extended to tackle other image-based tasks beyond classification, such as object detection with the YOLO model [58] or image segmentation with SegNet [59]. On top of supervised tasks, researchers are starting to leverage DNNs to generate images in an unsupervised fashion. This started with Autoencoder models [60], which were developed using a symmetric architecture with an objective to project input to a lower dimension before reconstruction of said input. While mainly used for dimensionality reduction at first, they became popular for image generation with the development of Variational Autoencoder (VAE) [30], which maps inputs to a distribution rather than a point as Autoencoder would. In parallel, Generative Adversarial Networks (GAN) [31] were developed on a different take. The idea of a GAN is to

train two DNNs jointly, with a generator tasked with generating images and a discriminator tasked with judging if an image is real or not. The two DNNs are "adversary" in the sense that to get better; the generator must fool the discriminator, which must instead find the real from the fake between authentic images of a dataset and the generated images of the generator.

Feedforward DNNs were first used for text data to train word embedding models with, for instance, Continuous Bags-Of-Words and Word2Vec models [61, 62]. Convolutional DNNs were also used in text data, using the convolutional operation to extract high-level features from a text embedding [63, 64]. Subsequently to Convolutional DNNs, Recurrent DNNs [65] and Long-Short Term Memory DNNs [66] were improved to tackle text data, leveraging their sequential power, which makes them a better fit than Convolutional DNNs to tackle the context dependencies of the language. This gave rise to the first DNNs used for the generation of text [67]. Young et al. [68] describe other applications of Recurrent and Convolutional DNNs to text.

Finally, with the rise of Transformers (see Chapter 2), DNNs such as ViT [69] or StableDiffusion [70] were developed to tackle image tasks such as classification or generation, while models such as BERT [71] and later ChatGPT [5] improved on text-based tasks.

3.2 Faults in Deep Neural Networks

As presented in Section 2.3.2, faults in DNNs can be sorted into two broad categories: specification-level and model-level. Here is an overview of works related to describing those faults in DNNs.

3.2.1 Specification-level

Several empirical research studies have analyzed bugs in DNNs and DL-based systems. Zhang et al. [72] published the first empirical study on real-world reproducible bugs occurring in 175 buggies TensorFlow-based DNN programs collected from StackOverflow and GiHub, aiming to identify their high-level root causes and symptoms. In particular, incorrect DNN parameters or structures, misuse of the application programming interface (API), and tensor shape issues were among the more prevalent faults. Then, Islam et al. [73] extended the investigated cases to include DNNs written using other DL frameworks such as PyTorch and Caffe, culminating in over 2700 posts from StackOverflow and 500 bug fix commits from GitHub to study the relationship and the evolution of different types of DL bugs, showing the same faults as Zhang et al. as most prevalent Humbatova et al. [74] proposed a comprehensive

taxonomy of real faults that occur in DL systems. Their main categories of faults encompass faults related to architecture (weights initialization, layer types, etc.), GPU usage (parallelism issues, wrong transfer to GPU), API calls (deprecated API, missing arguments, etc.), training related (data quality, optimization or loss function issue etc.) and tensors & inputs issues (tensor shape, input format etc.). The taxonomy was built using Stack Overflow discussions, GitHub issues/commits, and interviews with 20 researchers and practitioners. Moreover, they validated their taxonomy with an additional set of 21 researchers, concluding that most respondents experienced most faults in their taxonomy. Finally, Nikanjam et al. [75] studied faults specifically in Deep Reinforcement Learning programs covering, on top of the faults highlighted in previous studies, faults related to interaction and exploration of the environment and faults when updating the policy DNNs.

Faults can also happen within frameworks supporting such DNNs. Such bugs are particularly impactful as the faults will not originate within the users' programs but the framework underlying the programs. Jia et al. [76, 77] investigated symptoms, causes, and repair patterns of bugs inside TensorFlow as a typical DL library. They found that most faults occurred during the processing of variables (assignments, initialization, format, etc.), API calls (modified API due to change/update, etc.) or Corner Cases, i.e. particular use cases a function is not handling correctly. Similarly, Chen et al. [78] studied bugs at different levels (like User-level API and Graph-level implementation) within multiple DL frameworks. They found similar faults as Jia et al. (e.g. type issue, misconfiguration, or incorrect algorithm implementation). They showed that the Operation Implementation level (operations on the tensors) contains most of the faults with 30% of the analyzed bugs. Finally, as presented in Section 2.3.2, our work on silent bugs [37] within the Tensorflow/Keras framework explored their impact on users' DNN programs, showing notably over 50% of the silent bugs analyzed resulted in a high impact on the users' programs (modification of the computation or expected outcome etc.). Moreover, we validate our taxonomy of silent bugs through a survey of 103 practitioners, who highlighted that silent bugs have more impact on users' programs than traditional bugs.

3.2.2 Model-level

On the contrary to specification-level faults, model-level faults are harder to specify. As we described in Section 2.3.2, faults are generally caused by stochasticity of the learning process or underspecification due to limited data [79]. Several underlying phenomena can increase faults within models. For instance, "spurious correlations" [80, 81], that is, when variables are associated but not causally related, can introduce involuntary biases that induce faults

in the DNNs. For instance, a classifier trained to distinguish huskies and wolves, where all images of huskies are on a green background and pictures of wolves on a white background, will likely fail on images of huskies on a white background because of the relation within the data that the DNN might leverage inconspicuously leading to a fault. DNNs are also prone to "shortcut learning" [82], that is, learning an inner logic that will work well on training and similar independent and identically distributed sets but less so on corner-case inputs that are more distant. This is caused by the fact that DNNs generally will exploit features that minimize the loss of the available data without learning the intended underlying features. In that case, DNNs can lead to high performance using metrics such as accuracy on available data but will inevitably see their performance collapse on other data. The learning process is also vulnerable to learning non-robust features, that is, features that can easily be modified to modify the prediction of the DNNs [83]. Particularly, exploiting adversarial perturbations was shown to be an effective avenue for finding faults [12, 84, 85] in image based DNNs with faults at the model-levels likely originating from the way weights within DNNs were tuned during the training proces. For instance, Liu et al. [86] show it's possible, by changing slightly the value of a single bias value in a neuron, to induce a misprediction of the DNNs. LLMs suffer from similar faults, with the added faults linked to the transformer architecture, which would lead to hallucinations in practice. Ji et al. [36] give the main faults leading to hallucinations. For instance, encoders learning the wrong relations between words, which decoders can amplify. Transformers are also pre-trained on different tasks, such as predicting the next tokens based on a ground-truth, which can differ from their end goal, such as generating a text from an input partly generated by the transformers. This discrepancy can further add to potential faults. Finally, one main difference between Transformers and other DNNs is that the decoding strategy can employ some randomness where the predictions of DNNs are usually deterministic. For instance, temperature or nucleus sampling, which helps in the generation's diversity, correlates with a higher rate of hallucinations [87, 88].

3.3 Testing techniques for Deep Neural Networks

Testing DNNs is a crucial part of ensuring their safety. In a systematic literature review, we did [89] regarding certification of ML-based safety-critical systems; testing was one of the main aspects within the "Verification" pillars. Several literature reviews [13, 90, 91] listed existing testing approaches for DNNs, as well as limitations and future endeavours needed to enhance the testing of DNNs.

3.3.1 Testing metrics

Several testing metrics have been proposed to evaluate DNNs on any test set and/or benchmark [92–95]. In regression problems, Root Mean Square Error (RMSE) or Mean Square Error (MSE) are the most used., measuring the average deviation of the predictions compared to the ground truth values. In classification problems, Accuracy or the F1-score are generally used. Accuracy is the number of correct predictions compared to the number of total examples to predict. F1-score is the harmonic mean between Precision and Recall and measures the performance of one class at a time named the positive class. In that setting, positive predictions are predictions that were assigned the label of the positive class. Precision measures the ability of the model to identify positive examples correctly, while Recall measures the ability of the model to identify all positive examples. Measurements can also be modality-specific. For instance, in tasks dealing with images, one can use the Intersection Over Union in object detection or the Peak Signal-to-Noise Ratio and Inception Score [96] in image generation. The inception score quantifies the quality and diversity of a generated image through the marginal likelihood using an external classifier and is particularly used in GAN [31]. Many metrics were proposed regarding text data with, first, simpler ones, such as Exact Matching, which checks if the predicted text matches exactly the expected text, or the Levenshtein distance, which measures the difference between two sequences in terms of single-character edits. However, such metrics generally perform poorly, especially on open-ended tasks. In those cases, metrics using the structure of the language are more practical such as BLEU [38] uses the overlapping n-grams between a reference and a hypothesis, METEOR [97] (Metric for Evaluation of Translation with Explicit ORdering) which improves on BLEU and is based on the F1-score using a relaxed matching criterion (for instance, considering synonyms) of unigrams, ROUGE [98] (Recall-Oriented Understudy for Gisting Evaluation) which is a recall-based n-gram matching metrics contrary to BLEU which is precision-based. It's even possible to use trained external contextualized embeddings: using a pre-trained BERT (Bidirectional Encoder Representations from Transformers) [71] transformer, BERTr [99] score averages the recall score obtained overall tokens using the word embeddings obtained. Just like image data, some metrics are more tailored for sub-tasks, for instance, CodeBLEU [39] we introduced in Section 2.3.3. A widely used metric that prolonge Accuracy for LLMs is the Pass@k [100] metric, which was introduced mainly to account for the stochastic sampling that can be used in the inference of LLMs with, for instance, the temperature setting. In that setting, the output prediction of an LLM is not based on the next best tokens (commonly refer as greedy decoding) but based on a scaled distribution of the tokens based on the temperature factor. As such, the same input can yield multiple out-

puts, which helps get diverse answers. Pass@k is then defined for a benchmark as $\mathbb{E}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}]$ where, for each input in the benchmark, n is the number of samples generated, k a value such as $n \geq k$ and c is the number of correct samples. This metric gives the expected value of obtaining a correct answer across multiple generations. Each metric has pros and cons that should fit the particular use case to which they are applied. For instance, class-imbalanced problems should not rely on accuracy, while BLEU or METEOR are not proper metrics for dialogue systems. Moreover, while metrics can give a summary statistic of a particular task using available inputs, they might fail to provide more local insight on particular inputs or capture all nuances required.

3.3.2 Coverage Testing

A new set of DL-related criteria has been developed in the direct lineage of code coverage-related criteria. We saw in Chapter 2 NC [11] was the first criterion developed, directly inspired by traditional code coverage, to test the inner logic of DNNs. Simply put, NC designates the ratio of "fired" neurons (i.e., whose activation is positive/past a given threshold) on a whole set of inputs. The main criticism associated with this measure is that it is relatively easy to reach a high coverage without actually showing good resilience since it does not take into account relations between neurons as a pattern. Moreover, NC discards fine-grained considerations such as the level of activation by simply considering a boolean output. Following this first work, related criteria were developed to extend the definition and tried to tackle those issues. KMNC (K-Multisection Neuron Coverage) [101] was proposed to refine NC by discretizing the activations of neurons into buckets based on the values obtained on trained data. The coverage then consists of assessing how many of those buckets a test suite can cover. Top-k NC [102] tried instead to quantify how many top neurons were triggered by a test set. Top neurons are obtained by determining which neurons reach the highest activations on training data. T-way combinations [103] [104] looked at T-way combinations of neurons covered by a test set per layers, while Sign-Sign (and related) coverage [105] leveraged MC/DC (Modified Condition/Decision Coverage) criterion used in traditional testing. For instance, two neurons in two consecutive layers l_i and l_{i+1} are said to be sign-sign covered if there exist two test inputs x_1, x_2 such as both x_1 and x_2 trigger different signs on the activation of both neurons without leading to different signs on the activations of all other neurons in l_i. Intuitively, this shows that the sign change in one neuron independently affects the sign change in a consecutively connected neuron. Beyond coverage criteria based on neuron assessments, other criteria have been proposed. For instance, Kim et al. [106] proposed Surprise Adequacy, which quantifies how "surprising" an input is for a DNN, for instance,

using the distance between the vector of neuron activations of two inputs. Feng et al. [107] proposed a coverage criterion based on the Gini index, which quantifies how uncertain a DNN is of the prediction of a given test input, with the assumption that test inputs leading to a lower Gini score should lead to more faults.

3.3.3 Mutation Testing

MT is an established technique in Software Engineering [108, 109]. It has also been applied in settings where non-determinism is present, for instance, Probabilistic Finite Sates Machine [110]. Recently, researchers have been using MT to DNNs, with Nour et al. [111] evaluating MT effectiveness on DNNs, and DeepMutation [112], DeepMutation++ [113], or MuNN [114] proposing MT framework specific for DNNs targeted to supervised learning. These approaches notably distinguished between what they name *source-level* mutations, i.e. mutation acting on model *before training* (for instance, removing part of the training data), and *model-level* mutations, i.e. mutations acting on an *already trained* model (for instance, adding noise to weights of the model). When it comes to DRL, the only approach proposed was by Lu et al. [115], which strived to apply MT to DRL and studied how a well-crafted environment could be used to reveal a particular mutation, which is more akin to the fault detection method. However, DeepMutation and similar approaches do not necessarily consider the stochastic nature of DNNs and do not offer real faults-based mutation operators. Based on this observation, Jahangirova et al. [17] introduced a statistical version of the mutation test and compared empirically previous MT frameworks. This work was then further extended, leading to DeepCrime [116], where authors proposed a training set-based analysis of killability and redefined the notion of triviality using fuzzy logic. Nonetheless, those works are focused on supervised learning DNNs and do not account for all the stochastic of MT. There is still a debate on what constitutes an acceptable mutation in DNN. For instance, according to Panichella et al., [117], "source-level" mutations such as those evaluated in DeepCrime may not be regarded as mutations in the classical sense. The argument is that since DNN can be seen as a test-driven development procedure and the training data as a test suite, source-level mutation operators (for instance, removing a percentage of train data) affect the test suite rather than the production code. This interpretation is debatable since the training data is a crucial part of a DNN specification [118, 119] and not simply a part of a test suite. The training data is responsible for what the DNN "learns", contrary to simple test data that evaluates what the DNN has learned. Hence, mutating training data can be considered similar to mutating the production code since modifying the specification leads to a different model. In Chapter 4 and Chapter 5, similarly to previous works [17, 116], we consider mutations over the training process to be proper for MT. Finally, outside of MT

but on mutations, Shen et al. [120] used decision boundaries of Supervised Learning models to find a subset of the test set that is more likely to trigger given mutations.

3.3.4 Differential Testing

Classical testing methods for DNN classifiers require testing the prediction of an input against a ground truth value. If this is possible for a labelled dataset, it's much more complicated when no known labels exist. Here, the performance of the DNNs on the data distribution is trusted to predict new inputs in a relevant way. The existence of adversarial examples, inputs explicitly crafted to induce a failure on the DNNs or out-of-distribution examples, inputs which belong outside of the distribution the DNNs were trained on (e.g. a picture of a plane for a DNN trained only on animals) demonstrate that this strategy is limited, in particular with the absence of easily obtainable ground truth as highlighted with the oracle problem (see Section 2.3.4). We saw in Section 2.3.3 an example of a potential solution with N-versioning or metamorphic testing. On N-versioning, [121] showed the advantage of ensemble learning, which is based on N-versioning, against adversarial examples. In the case of pure N-versioning, NV-DNN [122] used this mechanism with a majority vote system to reject potential errors. The combination of models possible for N-versioning was also studied in more detail in [123], where the diversity of different architectures can bring over input error rejection/acceptance is explored. D2Nn [124] uses neurons more likely to contribute to errors to build a secondary network. The primary and secondary networks can then be used to compare predictions within a threshold. Yet, differential testing is not limited to semantically similar models, any semantic comparison allowing to build the "pseudo" oracle proxy is good. Hence, it's possible to use the specificity of DNN through neuron activations [125] to derive a "pseudo" oracle by gathering activation patterns of the train data. Another technique investigated in [126] used pairs of spatially/temporally similar images for the comparison. The main limitation is determining the policy and mechanism to balance false negative and positive examples correctly and the semantic modifications. In particular, the change must induce diversity while not being too different to preserve the similarity. On metamorphic testing, multiple approaches have been proposed for testing classifier DNNs [127], unsupervised DNNs [128] or even LLMs [129]. The main limitation is the reliance on the definition of metamorphic relations, that is, semantic preserving transformations, which can be complex and generally primarily involve manual work.

3.3.5 Test generation methods

Generally, the coverage criteria we listed earlier are not used as a plain testing metric like traditional software but rather as a way to generate test cases that maximize/minimize those criteria incrementally. As most DNNs are trained on image datasets, generating new images supporting increasing coverage is relatively simple and straightforward. To achieve this, techniques such as fuzzing are used to mutate base images sampled from a dataset randomly [102] [85] [130] or greedy search [131] and evolutionary algorithm [14] coupled with transformation properties can be used. Those properties are typically geometric and/or pixel-based transformations. More complex methods can be used, such as *concolic* testing [132], which mixed symbolic execution (linear programming or Lipschitz-based) with concrete input to generate new test cases helped by a heuristic based on coverage with improved results compared to other methods. Some techniques leverage GAN [84, 133] or assimilated to generate more realistic test images through coverage optimization. Of course, suppose the image obtained tends to be more "natural" than fuzzy or metamorphic ones. In that case, these methods suffer from the traditional downsides specific to GANs and need extra data. Besides coverage guide test generation, other methods rely on the particular objective of the paradigm tested to guide the generation. On image datasets, several techniques exist based on Adaptive Random test [134] leveraging Principal Component Analysis decomposition of network's features, techniques based on low-discrepancy among sequences of images and active learning [135], techniques using first-order loss [12], techniques coupled with metamorphic testing we presented earlier, either using entropy-based technique with softmax predictions [136] or searching critical image inputs following the metamorphic transform [137] proposed in metamorphic testing. Wicker et al. [138] used the Scale-invariant feature transform algorithm to identify salient parts of the image input and optimize for adversarial examples using a two-player game. Yaghoubi et al. [139] proposed a method to generate adversarial examples in a non-linear control system with a DNN in the loop by deriving a function from the interaction between the control and the DNN. In natural language processing applications, techniques such as TFAdjusted [140] or TextFooler [141] leverage perturbation of the text to generate new inputs. As for RL applications, simulation environments are used for training models. In this context, testing methods generate scenarios representing agents' behaviour in the environment that are more likely to lead to the failure of the learned policy. Meta-heuristic is the preferred method to generate procedural scenarios, whether it be evolutionary [142] or simulated annealing with covering arrays [143]. Scenario configuration can also be tackled from the point of view of a grammar-based descriptive system [144], which allows for a flexible comparison between cases. Gur et al. [145] proposed an agent that produces environments depending on the learning agent's skill level. Their approach allows for more methodolog-

ical training of agents and increases their robustness and ability to generalize. Cobbe et al. [146] introduced the Procgen benchmark, which comprises 16 procedurally generated environments that allow practitioners to test the generalization of the agents. Finally, Biagiola et al. [147] focused on generating test environments through a combination of binary/exponential search to map an agent's adaptation/anti-regression performance through the lens of continual learning. Aside from the choice of criteria or objective to follow, limiting factors of those techniques remain the time needed to obtain the generated test, the generalizability of the transformation and the adequate quality (or validity) of the test cases generated.

3.4 Limitations of Deep Neural Networks testing due to Deep Learning nature

Riccio et al. [13] identified several weaknesses of testing techniques applied to DNNs, namely hyper-parameter selection, failure/fault definition, realism and relevance of test cases, empirical methodology and metrics, non-determinism and computational cost. Our thesis focuses mainly on the last three weaknesses and, indirectly, on the definition of failure/fault. Regarding metrics limitations, several works have highlighted the different possible metrics, how and when to use them, as well as their limitations depending on the task to apply the DNNs on [8,94,148–150]. For instance, issues such as non-determinism, class imbalance, data importance, lack of uncertainty quantification (e.g. confidence interval) or domain-specific problems can bias the result of a metric [150, 151]. Moreover, the same set of metrics is still used in DL, for instance, accuracy (or any derivate) when assessing correctness, even though this metric can induce a bias when dealing with problems containing imbalanced classes. Even when dealing with coverage-related metrics we introduced in the previous section, several studies [18,152] showed the limitations of such metrics when it comes to helping in generating realistic inputs, finding actual faults within the models or offering reasonable computation time. Moreover, even if metrics are helpful as summary statistics, they are generally calculated over a given test set and can hide more local issues within the models. On non-determinism, studies [16,153,154] analyzed the impact of stochasticity on DNNs as well as described how to account for it in the evaluation process. Most testing techniques we described earlier deal with faults at the model-level but for an individual model. As such, the variable of interest is the particular DNN under test, so non-determinism is limited to the testing technique used. To mitigate stochasticity effects, multiple repetitions of the technique are generally done to show the resilience against stochasticity [155]. When it comes to model-level faults common for a group of DNNs, few testing techniques tackle this issue. For instance, while convolutional neural networks are vulnerable to adversarial attacks with few pixels being modified [156], testing techniques generally aim to find weaknesses inside a

targeted DNN [12, 85] and not weaknesses common within all convolutional DNNs of a same category or architecture. Similarly, in DRL, most studies aim to find faults in the agent under test [157]. Some studies [158, 159] analyzed the transferability of adversarial examples, which testing techniques use, between models. They highlight the relation between the transferability of adversarial examples and the transferability of knowledge. However, those studies mainly focus on specific types of adversarial examples and not the broad range of perturbations the previously mentioned testing techniques leverage. On LLMs, more studies have analyzed how different models behave either in terms of fault patterns in the code generated [158], type of hallucination produced [160, 161] or prompt modifications [162, 163]. On specification-level faults, previous approaches [112, 164, 165] did not necessarily account for the inherent stochasticity of DL on the process. Subsequently, Jahangirova et al. [17] highlighted the limitations which could cause specific faults not to be detected and new approaches such as DeepCrime [116] and DeepFD [166] accounted for this limitation. However, those approaches mainly focus on specifications of supervised learning DNNs. Finally, regarding the computation cost of testing techniques, most of them can be computationally expensive to apply, either in terms of time or memory [18]. To reduce this cost, several test reduction/selection methods [167–170] were developed. However, those approaches rely on a test set generated, at least partially, for the particular model under test, which, while reducing the cost, might still incur a high computation cost.

CHAPTER 4 MUTATION TESTING OF DEEP REINFORCEMENT LEARNING BASED ON REAL FAULTS

The objective of this chapter is to propose a reliable method for finding faults in DRL specifications while accounting for the different challenges of testing DL, specifically stochasticity. We use existing taxonomies of faults to build a set of 11 mutation operators relevant to RL. We also redefine MT to DRL by proposing two new detection criteria and using a simple heuristic to generate test cases for RL. This allows us to analyze the behaviour of the obtained mutation operators and their potential combinations (HOMs). We show that the newly proposed mutation detection definition as well as the generated test cases can improve in the number of mutations detected compared to the previous definition by up to 45%. Moreover, we found that even with a relatively small number of test cases and operators, we managed to generate HOMs with more than 50% being Subsuming (see Chapter 2.3.3), which can enhance testing capability in RL.

4.1 Introduction

Adapting existing tried and tested testing techniques such as MT, which we introduced in Chapter 2.3.3, to DL settings would greatly improve their potential verifiability. While some efforts have been made to extend MT to the Supervised Learning paradigm, little work has been done to extend it to RL. Applying MT in DRL has several questions we must tackle to apply it properly. Namely: What constitutes a test case in DRL? What mutations can be applied in DRL? How can a mutant be considered killed? And how do we account for the stochasticity issue when using MT? This chapter aims to answer those questions. Previously, Liu et al. [171] redefined how to detect a mutation in DRL by comparing the cumulative expected rewards of a mutated agent and a non-mutated agent and establishing an empirical threshold to declare a mutation detected. This, however, is limiting, and we show the issue as follows. Consider a DRL agent A_1 trained in an environment as described in Section 2.2.1. Said agent will have a trained policy network to take action given a state to maximize the expected cumulative reward. We expect that, if we were to create a mutant A_{M_1} of this agent, using whatever mutation is fit, we would end up with a different expected cumulative reward and a policy network which would take different actions in the same state. Given any object we consider to be test cases for DRL, we expect MT to allow us to detect the said mutation. However, say we instead had trained another agent A_2, using the same code and environment as A_1, and obtained a mutant A_{M_2} using the same mutation as before.

We may end up with different expected cumulative rewards and not detect the mutation in that case. Worse, the expected cumulative rewards might be the same, yet the policy networks differ in their actions. Thus, previous formulations of MT would fail to detect the mutation [114, 171]. Similarly, it would be possible that A_{M_1} obtain the same cumulative reward as A_1, the effect of the mutation being overcome by the training, and so the mutation is not detected. Indeed, as noted by Riccio et al. [13], for instance in classification, DNNs rarely manage no mispredictions on their test set, and even if they do, it is possible to craft inputs on which they might not produce a correct output. This implies that failures, and so faults, are inherent parts of DNNs. In that case, distinguishing a mutant A_{M_1} from a correct agent A_1 or A_2 might not be done using MT definition as we detailed previously because of the inherent stochasticity of DL models.

This study proposes a framework *RLMutation* for MT of DRL programs leveraging both FOMs and HOMs, see Section 2.3.3, adapted to DRL. We defined mutation operators for DRL motivated by existing taxonomized faults as well as new detection criteria. We then analyzed how effective the criteria are compared to the existing approaches on different DRL environments and algorithms by comparing the number of mutations detected. To leverage HOMs power to highlight more complex faults, we adapt existing work on HOMs [42] to the DRL task specificity. Thus, we aim to provide some insights into how MT could be applied to DRL.

The findings of this study highlight 11 mutation operators based on real taxonomized faults as well as a comparison of the impact of mutations detection definition design over the FOMs detected. We show the need to account for stochasticity of DL models, the definition of test environment as the "test cases" of DRL when using MT, and a heuristic to generate relevant test environments to study both FOMs and HOMs. The findings point out that, while MT applied to DRL raises numerous challenges to consider, from the mutation detection definition design to the generation of relevant test environments and HOMs, it is an effective venue to improve the testing of DRL models' reliability. As such, the findings emphasize the importance of focusing on established software testing methods to enhance the reliability of DRL.

To structure the study in this chapter, we define three RQs:

RQa_1: *What are the limitations of existing mutation detection definitions when applied to DRL?* Through this RQ, we aim to highlight the limitation of previously proposed criteria in DRL [171] and how to tackle the limitations by definition of new criteria. We particularly compare the number of mutations detected by each criterion.

RQa_2: *How are different agents and environments affected by the different mutations?* In this RQ, we aim to explore the mutations that can be detected among our sets of mutations leveraging the test environments we automatically generated. In particular, we show that patterns in which test environments detect certain mutations could be prolonged into a fault-localization technique.

RQa_3: *Do the HOMs generated from our FOMs possess the subsuming property similar to traditional software engineering?* Finally, in this RQ, we explore whether the defined criteria could be used to detect more complex mutations represented by HOMs, in particular *Subsuming* ones. We generate HOMs by coupling defined mutations and calculating the number of such HOMs we can detect.

4.2 Introductory example

As an introduction to MT in DRL, let's consider the following code snippet, which represents the "Missing Terminal State" mutation, one of the mutations we identified and used in our experiments.

```
class MutatedReplayBuffer(ReplayBuffer):

    ...

    def add(
            self,
            obs: np.ndarray,
            next_obs: np.ndarray,
            action: np.ndarray,
            reward: np.ndarray,
            done: np.ndarray,
            infos: List[Dict[str, Any]],
    ) -> None:

        self.observations[self.pos] = np.array(obs).copy()
        self.next_observations[self.pos] = np.array(next_obs).copy()
        self.actions[self.pos] = np.array(action).copy()
        self.rewards[self.pos] = np.array(reward).copy()

        self.dones[self.pos] = 0 # Should have been np.array(done).copy()
```

Listing 4.1 An example of a mutation with the *Missing Terminal State* mutation.

In DRL environments, the terminal state is defined as the last state the agent transitioned

to before the environment was terminated. The termination criteria for an environment can be reaching the goal state, stepping into a trap state, or reaching a time limit. Usually, in designing a DRL environment, the terminal state contains a different reward signal than the rest of the transitions, e.g., if the agent falls into a trap, it should receive a negative reward. This mutation operator simulates cases where the developer incorrectly implements identifying the termination criteria. As a result, the agent will not receive the termination signal and cannot correctly determine the correlation between the actions taken and the results achieved. In that case, the mutation affects the "dones" array, which stores whether or not the current state is a terminal state and so should end an episode. Instead of storing whether or not the current state is terminal, it will store "0" no matter what, that is not a terminal state. This mutation was inspired by actual faults found in DRL practitioners and developers. Note that the fault might not actually be as trivial in the code (i.e., the developer putting a 0), but the effect of the fault would be the same (i.e., the terminal state is never reached). When applying MT as defined previously, finding this fault is not trivial. Indeed, while their terminal state is never reached, an episode will also end when a certain number of steps is reached (Section 2.2.1) and so the agent can still "learn", even if imperfectly, and reach a similar cumulative expected reward as a non-mutated agent. As we detailed in the introduction, the mutation effect might not be perceived on the cumulative expected reward, so the mutation would not be detected.

4.3 Methodology

This section presents the main components of MT (detection criterion, mutations and test cases) and how to apply it in DRL.

4.3.1 Mutation Testing criterion

As we described earlier, Lu et al. [171] proposed an approach to evaluate a crafted environment's ability to reveal mutants based on mutation operators designed for DRL. They defined a mutant as killed if, for a healthy agent A and a mutated agent A_M, the ratio p_M/p of their average rewards over n episodes on a given environment E are inferior to a given threshold θ. In the following, we will refer to this criterion as AVG (for average). However, as explained in the introduction and previous section, using this criterion there can be cases where (A_1,A_{M_1}) would reveal the mutation, yet (A_2,A_{M_2}) would not because of DRL's inherent stochasticity, which can deeply change the results among trained agents using the same environment and hyperparameter configurations but with different seeds [172, 173]. Hence, two users training two agents on the same specification would have two different test results.

To mitigate this issue, we propose to adopt a criterion that was developed in MT for Supervised Learning: in DeepCrime [116], the authors proposed to consider MT through the lens of statistical testing over the accuracy of a distribution of instances that compares n non-mutant instances against n mutant instances. In particular, for a given test set $\mathcal{T}$ and the sets of accuracy over $\mathcal{T}$ of non-mutated models $(A_{N_1}, ..., A_{N_n})$ and mutated models with a given mutation operator $(A_{M_1}, ..., A_{M_n})$, they defined the following test function:

$$MT_{\mathcal{T}} = \begin{cases} 1 & \text{if p-value} < 0.05 \text{ and effectSize} \geq 0.5 \\ 0 & \text{else} \end{cases} \tag{4.1}$$

where the *p-value* is obtained by using *Generalised Linear Model* [174] and the *effectSize* is calculated using *Cohen's d* [175]. While not directly part of the test function, they consider the power analysis to exclude mutations for which the test's statistical power is too low (with the threshold $\beta \geq 0.8$).

In the DRL case, we modified DeepCrime's criterion as follows: First, as test sets do not exist in DRL, we will instead rely on the environments the agents are trained on. Those test environments will be what MT assess. This already raises a question as we only have one test environment. In Section 4.3.4 of this chapter, we will detail one approach to generate test environments in a relatively simple yet meaningful way. Secondly, as accuracy is not a DRL metric, we will instead rely on the agent's expected cumulative reward at the end of the n episodes in a given environment. This approach will be referred to as R (for reward) in the following.

However, from the statistical definition above, while some mutated agents can exhibit a lower reward than some healthy agents, both healthy and mutated agents' reward distribution might not exhibit a statistically significant difference. This can particularly be the case when a few agents end up exhibiting a very different reward because of the effect of the mutation not being overcome by the training or being accentuated by a non-favourable initialization, as the initial seed can have a high impact on the training [172, 173]. Removing such data points is not ideal as it masks potentially helpful information. One way to account for those data is to leverage the fact that we know the variation between healthy/mutated agents but also *between* healthy agents. Therefore, any potential outlier would lead to a significant difference between them. Such variation can be estimated by calculating the Hellinger distance [176] of the rewards between samples of the healthy agents' distribution (*intra* distance) and the same distance between samples of the healthy/mutated agents distribution (*inter* distance). The Hellinger distance is a metric bounded between 0 and 1, with 0 meaning both distributions are the same, which thus can be interpreted as a measure of similarity between the distributions.

The discrete case is defined for two distributions, P and Q, as:

$$H(P,Q) = \frac{1}{\sqrt{2}}||P - Q||_2 \tag{4.2}$$

where $||.||_2$ is the L^2 norm.

By repeating the calculation through sampling from both agents' reward distribution, one can obtain the distributions of the *inter/intra* distance and use the same statistical test used in the definition *R*. We refer to this criterion as *DtR* (for Distance to Reward).

4.3.2 Mutation Operators for Deep Reinforcement Learning

Lu et al. introduced several mutation operators applicable to DRL [171]. Nonetheless, their operators present several shortcomings. First, some can not be generalized to any DRL algorithm which limits their relevance, for instance, mutations based on the *epsilon* parameter are limited to a subset *Off-policy* algorithms such as *DQN*. Secondly, some operators, such as removing/adding a neuron on a particular layer, will most likely lead to a crash, as the fault itself leads to cascading changes in the model architecture, as pointed out by Humbatova et al. [116]. Finally, some of the mutation operators are not justified since they are not based on real faults which undermines the usefulness of the operator, for instance, *shuffling the replay priority* in the replay buffer is not motivated by any real fault.

Thus, to obtain relevant FOMs, we started from operators defined in [171] as a basis and further extracted existing taxonomies reporting on bugs affecting DRL or Deep Learning models [75] [74] or directly adapting existing mutation operators [17] [116] used in Supervised Learning. We obtained a (non-exhaustive) list of mutation operators that we divided into three categories (i.e., environment, agent, and policy) based on what the mutation is affecting. Due to space constraints, we only briefly describe the mutation operators in this section. A comprehensive description can be found in our replication package [177] referencing the specifics that motivated each operator along with a comparison with operators defined in [171].

Environment-level: The environment-level mutations are meant to simulate the faults that can happen when an agent receives observations as it interacts with the environment, i.e. receiving an incorrect observation. These faults can, for instance, be the results of faulty sensors, faults in environment design, or even nefarious attacks [178, 179]. Each operator is also likely to be applied to a given step, with 100% probability, meaning that the mutation is applied to every step of the agent's training.

- *Reward Noise (RN)*: Adds a (Gaussian) noise to the true reward that the agent was meant to receive and returns it to the agent.
- *Mangled (M)*: Damages the correlation between collected experiences. This operator returns a random s_{t+1} and r_t, not the state and reward the agent should receive according to its current state and the action taken.
- *Random (Ra)*: Similar to the mangled mutation operator, the *Ra* mutation returns a $(s_t, a_t, r'_t, s'_{t+1})$ tuple to the agent. However, unlike the *M* operator where s'_{t+1} and r'_t are selected randomly and are not associated with each other, the random operator returns some s'_{t+1} and r'_t to the agent which was sampled from the same experience tuples but has no association with s_t and a_t.
- *Repeat (Re)*: Returns the previous observation to the agent. If the agent has two consecutive experiences in the form of (s_t, a_t, r_t, s_{t+1}) and $(s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$, this operator returns $(s_{t+1}, a_{t+1}, r_t, s_{t+1})$.

Agent-level: As shown in [75], many of the issues developers face in implementing DRL algorithms result from incorrect DRL concept coding. The agent-level mutations are meant to simulate the faults that can happen when a developer makes mistakes in implementing the concepts of a DRL-based agent in code.

- *No Discount Factor (NDF)*: Removes the discount factor γ from the reward calculation.
- *Missing Terminal State (MTS)*: Removes the terminal state of an episode (i.e., reaching the goal or falling into a trap, etc.) and so the correlation between the actions taken and the results achieved.
- *No Reverse (NR)*: Wrongly reverses the order of the received rewards, making the agent learn an incorrect association between the experiences.
- *Missing State Update (MSU)*: Removes the state update after the agent takes an action, meaning the agent will always see the same state in the experience tuple e.g., $(s_t, a_{t+1}, r_{t+1}, s_{t+2})$ during training.
- *Incorrect Loss Function (ILF)*: Modifies the loss function used in the DNN that learns the policy.

Policy-level: This category contains mutations affecting the agent's policy. In general, as the policy is implemented via DNNs, these mutations are similar to those previously defined in Supervised Learning [116].

- *Policy Activation Change (PAC)*: Changes the default activation function used in the policy network of the agent.
- *Policy Optimizer Change (POC)*: Modifies the default optimizer of the algorithm while keeping the original learning rate.

4.3.3 Higher Order Mutations

Based on our previously defined FOMs, we also evaluated HOMs in DRL. To identify interesting HOMs, we follow a similar procedure to what was introduced in Jia et al. [42], presented in Algorithm 1.

Algorithm 1: HOM generation algorithm

Input : m FOM $\mathcal{F} = FOM_1, ..., FOM_m$, n healthy agents $\mathcal{A} = A_1, ..., A_n$, for each FOM the relevant mutated agents $\mathcal{A}_{FOM_i} = A_{1,FOM_i}, ..., A_{n,FOM_i}$, initial environment E_0, parameters to modify *params*
Output: The set of FOMs $\mathcal{F}^*$ to consider for the HOM

```
1  E = {E_0, ..., E_p} ← GenerateBoundsEnvironments (A, params, E_0);
2  F* ← ∅;
3  foreach f ∈ F do
4  |   i ← 0;
5  |   foreach e ∈ E do
6  |   |   if IsDifferent (A, e, A_f, e) then
7  |   |   |   i++;
8  |   end
9  |   if i ≠ |E| and i ≠ 0 then
10 |   |   F* ← F* ∪ f
11 end
```

Starting from Line 2, the algorithm describes how we implement, in our case, the heuristic of [42] to determine which FOMs to consider for the HOMs. We aim to determine which FOMs are not trivial, i.e., not detected by all test environments or none. To evaluate if an environment detects a mutation, we use one of the MT criterion described earlier, using the rewards of healthy agents $\mathcal{A}$ evaluated on environment e and mutated agents $\mathcal{A}_f$ evaluated on the same environment (function $IsDifferent$ in Algorithm 1). However, different test environments are needed to apply the heuristic. As such, we need a simple yet effective way of generating more test environments.

4.3.4 Generating test environments

One way to generate new test environments relatively straightforwardly that can be automated is to modify certain properties (i.e., parameters) of said environments. As such, we define a test environment E_i as dependent on its physical parameters. For instance, the CartPole environment [180] consists of a cart with a pole connected to it through a pivot. Two parameters of the environment are the cart's and pole's masses, which can be varied and then influence the agent's decision, which can reveal faults.

Algorithm 2: Generate Bounds Environments

Input : n healthy agents $\mathcal{A} = A_1, ..., A_n$, initial environment E_0, parameters to modify *params*
Output: The set of boundary environments $\mathcal{E}$

$\mathcal{E} \leftarrow \{\}$;
foreach $p \in params$ **do**
  `// Test environments on the upper boundaries`
  if $E_0.p \neq p.l_{upper}$ **then**
    $E_c \leftarrow E_0$;
    $E_c.p \leftarrow p.l_{upper}$;
    if *not* `IsDifferent` *($\mathcal{A}$, E_c, $\mathcal{A}$, E_0)* **then**
      $\mathcal{E} \leftarrow \mathcal{E} \cup E_c$;
    else
      $E_b \leftarrow E_0$;
      while `CheckPrecision` *(E_c, E_b)* **do**
        $p_m \leftarrow \frac{E_b.p + E_c.p}{2}$;
        if *not* `IsDifferent` *($\mathcal{A}$, E_c, $\mathcal{A}$, E_0)* **then**
          $E_b.p = p_m$;
        else
          $E_c.p = p_m$;
        end
      end
      $\mathcal{E} \leftarrow \mathcal{E} \cup E_b$;
    end
  else
    $\mathcal{E} \leftarrow \mathcal{E} \cup E_0$;
  end
end
...;
`/* Similar for lower boundaries */`
...;
for $i \leftarrow 1$ **to** *depth* **do**
  $\mathcal{E}_m \leftarrow \{\}$;
  for $j \leftarrow 1$ **to** $|\mathcal{E}|$*-1* **do**
    if $\mathcal{E}[j] \neq E_0$ *AND* $\mathcal{E}[j+1] \neq E_0$ **then**
      $E_c \leftarrow \frac{\mathcal{E}[j].p + \mathcal{E}[j+1].p}{2}$;
      if *not* `IsDifferent` *($\mathcal{A}$, E_c, $\mathcal{A}$, E_0)* **then**
        $\mathcal{E}_{\Updownarrow} \leftarrow \mathcal{E}_m \cup \{\mathcal{E}[j], E_c, \mathcal{E}[j+1]\}$;
      else
        $E_b \leftarrow E_0$;
        while `CheckPrecision` *(E_c, E_b)* **do**
          $p_m \leftarrow \frac{E_b.p + E_c.p}{2}$;
          if *not* `IsDifferent` *($\mathcal{A}$, E_c, $\mathcal{A}$, E_0)* **then**
            $E_b.p = p_m$;
          else
            $E_c.p = p_m$;
          end
        end
        $\mathcal{E}_m \leftarrow \mathcal{E}_m \cup \{\mathcal{E}[j], E_b, \mathcal{E}[j+1]\}$;
      end
  end
  $\mathcal{E} \leftarrow \mathcal{E}_m$;
end
if $E_0 \notin \mathcal{E}$ **then**
  $\mathcal{E} \leftarrow \mathcal{E} \cup E_0$;

Finding diverse and non-trivial test cases that can impact the FOMs differently is not obvious. Indeed, an exhaustive search is impractical because of the high number of parameter combinations and the computational cost of evaluating the agent's behaviour, mutated or not, in each candidate test environment. At the same time, a random search might lead to many trivial test environments and similarly can be computationally expensive. Thus, leveraging some form of heuristic, even simplistic, is needed. Intuitively, environments close to the

initial one have a high chance of yielding the same decision concerning the mutation, which would limit the relevance of such environments. At the same time, going too far away from the initial environment might unexpectedly affect the behaviour of any agent and will also lower the relevance of the test environment. Consequently, finding the right balance between the two extremes is needed. This, however, assumes a certain continuity of the parameter space. Nonetheless, other works using a similar method such as Biagiola et al. [147] suggest such an approach can be used.

Therefore, we propose the following: by using only healthy agents to decrease the computational cost, we can look for "frontier" environments, that is, environments that are different enough from the initial environment in terms of behaviour but not too different. One way to find such a tipping point is to leverage binary search between healthy agents' reward on the initial environment and agents' reward on the generated candidate test environment, comparing them with the previously defined mutation detection definition (see Section 4.3.3). This is the goal of the function $GenerateBoundsEnvironments$ on Line 1 of Algorithm 1 and presented in Algorithm 2.

The method first looks for test environments along the parameter axes by applying a binary search over only one parameter between the initial environment (E_0) parameters and the defined search limits of the parameter $p.l$ (Line 2-25 Algorithm 2). In each step of the search, for the healthy agent, the distribution of reward obtained on environment E_c is then tested against the distribution obtained on the initial environment E_0 (function $IsDifferent$ in Algorithm 2). The search stops when a certain precision is reached between the lower/upper boundaries of the environment's parameters. Then (Line 26-46), the algorithm will loop for a certain number of pre-defined *depths*. It searches for test environments *in-between* the previously calculated environments by iteratively applying the same method. Note that the algorithm was implemented for a 2-D space since it was used in our experiments, yet it can be easily extended without losing generality to n-D. Thus, for the 2-D case, from 4 points, the algorithm will yield 8 after the first *depth* and 16 after the second. In the end, the set of test environments is returned in Algorithm 1. A representation of the algorithm process is also given in Figure 4.1.

4.4 Experimental design

This section describes the implementation of studied environments and introduces our RQs. Furthermore, we describe our DRL training process, experimental design, mutation detection definitions used, and obtained results.

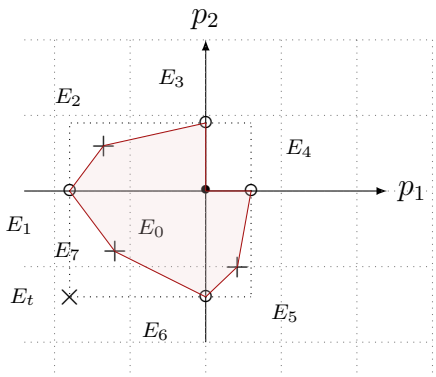


Figure 4.1 Generated test environments along the heuristic-based frontier (in red) on which healthy agents' reward distribution differs too much from the initial environment one.

4.4.1 Implementation and models

We used Python 3.8 and Stable Baselines 3 [181] framework version 1.6.2 as a basis to implement our RL models and mutations. The motivation to use this framework is double: first, it allows us to evaluate some mutations that could directly affect the users of such a framework, for instance, a wrong activation or optimizer for the policy network. Secondly, it serves as a solid basis for implementing mutations affecting the potential customized code of the user. Indeed, mutation can be introduced this way by overriding the base code by modifying only the relevant part, which ensures our code is less prone to unintended errors and allows better control over the mutation implementation.

To test our mutations, we chose two well-known environments in the RL community: *CartPole* and *LunarLander*. The CartPole environment [180] consists of a cart with a pole connected to it through a pivot. The agent aims to stabilize the pole and prevent it from falling by applying an appropriate amount of horizontal force by moving the cart left or right. LunarLander is a more complex environment representing a spaceship that must land on a surface delimited by two flags. The agent can control the spaceship by throttling it in three directions (left, right, and down) [182].

We used three deep RL algorithms: similarly to the previous paper [171], we use Deep Q-Network (DQN) [183]. On top of that, instead of using plain Q-learning, which is a relatively simplistic algorithm, we will consider two other algorithms, namely Advantage Actor-Critics (A2C) and Proximal Policy Optimization (PPO), which are classical algorithms in deep RL. This allows us to compare *Off-Policy* algorithms (DQN) with *On-Policy* algorithms (PPO, A2C), which are two major approaches in RL. We trained for each environment-algorithm-mutation, $N = 20$ agents with different seeds. For each algorithm/environment, we report

the average accumulated reward and standard deviation across the agents in Table 4.1. We also remind the readers of the mutations used in Table 4.2.

Table 4.1 Average rewards across the 20 agents with the standard deviation in parenthesis.

	PPO	**A2C**	**DQN**
CartPole	500 (0)	500 (0)	414 (143)
LunarLander	262 (16)	141 (56)	155 (84)

Table 4.2 Mutation operators summary

Category	Operator	Description
Environment-level	Reward noise (RN)	Adding noise to the reward the agent receives
	Mangled (M)	Returning next state and reward which are not related to each other
	Random (Ra)	Returning next state and reward that are related to each other but not related to the action taken
	Repeat (Re)	Returning next state and reward from previous observation
Agent-level	No discount factor (NDF)	No discount factor during calculating cumulative rewards
	No reverse (NR)	Not reversing the order of the received rewards during calculating cumulative rewards
	Missing state update (MSU)	Not updating agent's observations
	Missing terminal state (MTS)	Failing to save the terminal state observation
	Incorrect loss function (ILF)	Defining an incorrect loss function (wrong formula, etc.)
Policy-level	Policy activation change (PAC)	Different activation for agent's neural network
	Policy optimizer change (POC)	Different optimizer for agent's neural network

Some mutations rely on some parameters, which will be defined with both the mutation op-

erator identifier and the parameter. For instance, *Policy Activation Change* requires defining which new activation will be used, such as Stochastic Gradient Descent (SGD), and so will be noted as PAC_SGD. Environment-level mutations are based on some probability of being applied for each given step. So $M\_1.0$ means that the mutation operator *Mangled* will be used with a 100% probability at each step. If no parameters are needed, then just the identifier is used. For instance, *No Reverse* will simply be noted as NR. All mutations are used for all agents, except NR and $PAC\text{-}ReLU$ for DQN, the first one not applicable to DQN while the second is the default activation of DQN.

4.4.2 Research Questions

We first remind the RQs that will be used to guide our study:

RQa_1. Limitations of current mutation detection definitions

As we detailed in Section 4.3.1, MT for DRL can be applied in at least three ways. The first definition (AVG) is proposed in [171], where a mutation is considered detected if the ratio of the average rewards over n episodes gained by the healthy agent to the average of the mutated one is lower than a threshold θ. The second definition (R) is a distribution-wise statistical test which, in DRL's case, can be based on the reward over n episodes instead of the accuracy over the test set. The third one (DtR) improves the second definition using the distance between inter/intra rewards instead of simply the rewards. **This question aims to evaluate the relevance of those mutation detection definitions in DRL over the mutation operators we defined previously.**

To compare the definitions fairly, we will need to modify the first approach (ratio of the average rewards over n episodes gained by the healthy agent to the average of the mutated one) as it does not account for N multiple agents. We will count the number of times the ratio is lower than a certain threshold among N agents trained independently. The choice of the threshold θ can impact the number of mutations found, as the higher the threshold is, the more likely it is that a mutation can be found. Yet, at the same time, a too high θ might reduce the usefulness of the ratio in the case a mutated agent behaves similarly to a healthy agent in the test environment, e.g. an agent recovering from the mutation. Lu et al. [171] chose a threshold of $\theta = 0.8$. Instead, we chose a more conservative value of 0.9, making the test more likely to find a mutation. We will consider the mutation detected if at least 80% of the N ratios are lower than the 0.9. The second and third methods will use a statistical test over the reward of N agents similar to the original implementation in DeepCrime [116]. On top of that, the third method will use samples of half of the agents available to calculate

the distance using the Hellinger distance as described in Section 4.3.1.

RQa_2. Behaviours of FOMs on generated test environments

The goal of this RQ is to investigate the behaviours of FOM concerning the type of mutations used and the generated test environments. Test environments were generated following the procedure detailed in Section 4.3.2 with a depth of 1. Depth was chosen to keep a low number of environments so that computation would not be too expensive. Increasing it would increase the number of environments generated and, likely, the potential number of relevant FOMs at the cost of increased processing time. The procedure allows us to **obtain a finer grain analysis of the FOM operators** by getting the number of test environments detecting a given FOM. Moreover, it is then possible to analyze the parameters of the test environments in order **to assess what parameters' set is more likely to trigger a certain mutation** (for instance, for CartPole, if some mutations are more likely to be triggered when the mass of the cart is lowered or not). Finally, we will leverage FOMs, as presented in Section 4.3.2, to **deduce the interesting FOMs to generate potential subsuming HOMs**.

The test environments will be generated by altering two parameters of each environment: for CartPole, the *mass of the cart and the pole* will be modified, while for LunarLander, the *gravity* and *the side engine power* of the spacecraft will be modified. We refer the readers to our replication package [177] for the initial parameters and the search boundaries used.

RQa_3. Properties of generated HOMs

Finally, similar to RQa_2, in this RQ, we aim to **analyze the HOMs generated** in the previous experiments by reusing our test environments. The goal is to understand which property the said generated HOM possesses, following the description briefly presented in Section 2.3.3. In particular, **we aim to see if we can generate Subsuming HOMs** from our defined FOMs and test environments.

Using the chosen FOMs from RQa_2, we generate HOMs that will, in turn, be used to train N mutated agents for each environment/algorithm/mutation operator. We will then evaluate them on the previously generated test environments and compare the results obtained from RQa_2's FOMs to deduce their properties.

4.5 Results

This section presents the experimental results obtained while answering our three RQs.

4.5.1 RQa_1. Existing limitations of current mutation detection definitions

Table 4.3 FOMs test results. ✓ means mutation is detected, while ✗ means mutation is not detected or the test's statistical power is too low (inconclusive). The detection criteria used are **AVG**: Average Reward Mutant/Healthy with the value in parentheses being the proportion of ratios below the threshold θ, **R**: Reward-based statistical test, and **DtR**: Distance to Healthy Reward statistical test. "**-**" means mutation is not applicable.

Environment	DRL Algorithm	Detection Criteria	Mutations											
			ILF	M-1.0	Re-1.0	Ra-1.0	RN-1.0	NDF	NR	MSU	MTS	PAC-ReLU	PAC-Sigmoid	POC-SGD
CartPole	PPO	AVG	✓ (1.0)	✓ (1.0)	✗ (0.0)	✓ (0.95)	✗ (0.0)	✗ (0.65)	✓ (1.0)	✓ (1.0)	✗ (0.05)	✗ (0.05)	✗ (0.25)	✓ (1.0)
		R	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✓
		DtR	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
	A2C	AVG	✓ (1.0)	✓ (0.95)	✗ (0.15)	✓ (1.0)	✗ (0.25)	✗ (0.35)	✓ (0.9)	✓ (1.0)	✗ (0.05)	✓ (1.0)	✗ (0.2)	✓ (1.0)
		R	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓
		DtR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	DQN	AVG	✓ (1.0)	✓ (1.0)	✓ (0.9)	✓ (1.0)	✗ (0.3)	✓ (1.0)	-	✓ (1.0)	✓ (1.0)	-	✓ (0.8)	✓ (1.0)
		R	✓	✓	✓	✓	✗	✓	-	✓	✓	-	✓	✓
		DtR	✓	✓	✓	✓	✗	✓	-	✓	✓	-	✓	✓
LunarLander	PPO	AVG	✓ (1.0)	✓ (1.0)	✗ (0.2)	✓ (1.0)	✗ (0.05)	✓ (0.95)	✓ (1.0)	✓ (1.0)	✗ (0.05)	✗ (0.05)	✓ (0.95)	✓ (1.0)
		R	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✓
		DtR	✓	✓	✗	✓	✗	✓	✓	✓	✓	✗	✓	✓
	A2C	AVG	✓ (1.0)	✓ (1.0)	✗ (0.3)	✓ (1.0)	✗ (0.45)	✗ (0.45)	✓ (0.85)	✓ (1.0)	✗ (0.45)	✗ (0.2)	✗ (0.6)	✓ (1.0)
		R	✓	✓	✗	✓	✗	✗	✓	✓	✗	✓	✗	✓
		DtR	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓
	DQN	AVG	✓ (0.95)	✓ (0.95)	✓ (0.9)	✓ (0.95)	✗ (0.6)	✓ (0.95)	-	✓ (0.95)	✗ (0.7)	-	✓ (0.8)	✓ (0.95)
		R	✓	✓	✓	✓	✗	✓	-	✓	✗	-	✗	✓
		DtR	✓	✓	✓	✓	✗	✓	-	✓	✗	-	✓	✓

Results of FOMs for a given algorithm/environment and detection definition are given in Table 4.3 with *AVG* being the method using the average of the ratio and *R* being the method using reward-wise statistical comparison. As one can see, the limitations of *AVG* we briefly introduced in Section 4.3.1 are shown: for multiple mutation operators, no matter the environment/algorithm, the agents evaluated by the ratio might yield a very different result. If some mutations such as *ILF* or *POC-SGD* seem to be detected by the test environment on all healthy/mutated pairs, many mutations show a relatively low number of pairs declaring them detected. It is even possible that, for mutations with relatively small effects, the mutated agent will have a higher reward than the healthy agent if the initialization was not favourable for a particular healthy agent. As such, it seems that the stochasticity of the training process greatly influences the ratio obtained, leading to mutation not being detected across multiple test runs. On the other hand, using a reward distribution statistical test (*R*) leads to more mutations being detected in all cases except for *LunarLander/DQN*. In particular, this method allows mutations to be detected while *AVG* only had a handful of agent pairs declaring that mutations were detected using the same test environment. For instance, *CartPole/A2C/RN-1.0* where only 25% of ratios revealed the mutation.

RQa_{1-1} Previously introduced mutation detection criterion in DRL based on a ratio between healthy/mutated agents is not sufficient as it might miss a high number of mutations because of the stochasticity of the training process, particularly for mutations with low effect on the training which are harder to find. Using distribution-wise statistical tests based on the reward improves this aspect.

Nonetheless, for the same test environment, the distribution-wise test R still does not allow the detection of a high number of mutations, even mutations for which there are a sizable amount of ratios declaring the mutation detected with the *AVG* definition, such as *LunarLander/A2C/NDF*. The *DtR* criterion might be better suited in those cases. The results are presented in Table 4.3 in the *DtR* rows.

This approach improves over simply using the rewards of healthy/mutated agents. In particular, it manages to lead to some mutations with relatively low *AVG* scores to be detected by the same test environment, such as *LunarLander/PPO/MTS*, as *DtR* allows to account for the previous potential outlier problem. All in all, it is possible to improve the amount of mutations detected by up to 45% with the *DtR* criterion compared to the criterion defined previously. Nonetheless, one can see the method is not perfect as mutations such as *LunarLander/DQN/RN* end up not being detected while a sizable number of agent pairs are declared mutant by *AVG*. In those cases, the variation among healthy agents might be too pronounced compared to the ones between healthy and mutated agents, which is why *DtR* can not catch the mutation. This, however, highlights an important observation: the choice of the mutation detection criterion design in DRL is a crucial step to determine if a mutation is detected or not.

RQa_{1-2} More than choosing which mutation operators to consider, careful selection of *how* to use MT when applied to RL is another important parameter. For instance, using Hellinger distance to healthy reward instead of plain reward allowed for more mutations being detected for the same test environment. Thus, investigating different mutation detection criterion designs in DRL is crucial.

4.5.2 RQa_2. Behaviors of FOM on generated test environments

Following Section 4.3.4, we generated several test environments for each algorithm/environment and made several observations. Results are presented in Table 4.4. Mutations flagged in green for a given environment/algorithm mean that the mutation is not trivial and will be relevant when we need to generate HOMs. Note that we focus only on distribution-wise

Table 4.4 Number of test environments detecting each FOM. Green Cells are FOM that will be used to generate HOM.

Environment	DRL Algorithm	Detection Criteria	Mutations											
			ILF	M-1.0	Re-1.0	Ra-1.0	RN-1.0	NDF	NR	MSU	MTS	PAC-ReLU	PAC-Sigmoid	POC-SGD
CartPole	PPO	R	4/4	4/4	0/4	4/4	3/4	4/4	4/4	4/4	3/4	3/4	4/4	4/4
		DtR	4/4	4/4	3/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
	A2C	R	4/4	4/4	0/4	4/4	4/4	3/4	4/4	4/4	3/4	4/4	2/4	4/4
		DtR	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
	DQN	R	4/4	4/4	4/4	4/4	0/4	4/4	-	4/4	4/4	-	4/4	4/4
		DtR	4/4	4/4	4/4	4/4	3/4	4/4	-	4/4	4/4	-	4/4	4/4
LunarLander	PPO	R	8/8	8/8	0/8	8/8	1/8	8/8	8/8	8/8	0/8	4/8	8/8	8/8
		DtR	6/6	6/6	2/6	6/6	3/6	6/6	6/6	6/6	3/6	3/6	6/6	6/6
	A2C	R	9/9	9/9	0/9	9/9	0/9	6/9	9/9	9/9	2/9	3/9	5/9	9/9
		DtR	9/9	9/9	2/9	9/9	4/9	7/9	9/9	9/9	4/9	7/9	6/9	9/9
	DQN	R	9/9	9/9	8/9	9/9	0/9	9/9	-	9/9	0/9	-	0/9	9/9
		DtR	9/9	9/9	9/9	9/9	4/9	9/9	-	9/9	6/9	-	4/9	9/9

tests from this point on, as RQa_1 illustrated the drawback of the mutation detection criterion in [171].

The first observation we make is that five mutations are detected by all generated test environments, namely *ILF*, *MSU*, *NR*, *POC-SGD*, *M-1.0* and *Ra-1.0*. Those mutations have too much of an impact not to be detected, which is also highlighted in Table 4.3, as those mutations yielded a score > 0.85 when the initial test environment was used with the ratio method (*AVG*). In those cases, the stochasticity is masked by the high impact of the mutations, and all trained agents behave similarly. Thus, generating multiple test environments allowed us to see that those mutations might not be much of interest. Indeed, they are rather trivial to detect and so can probably be ignored.

Secondly, we see the test environments generated on *CartPole* are relatively more likely to catch mutations, with most of the mutations being detected by at least half the test environments compared to the ones generated on *LunarLander*. While the number of test environments generated might play a role, it is likely because *LunarLander* is a more complex environment than *CartPole*, so mutations might become harder to detect as the environment complexity increases. For instance, *CartPole* seems to be less sensitive to the *MTS* or *PAC* mutations, i.e., removing the terminal state of an episode or changing the activation function of the policy network does not seem to affect much the agents trained on *CartPole* contrary to *LunarLander*.

RQa_{2-1} Contrary to using only the initial test environment, generating additional test environments allows us to roughly evaluate which mutations might be trivial and which are more interesting based on the number of environments detecting them. We also found that it appears that the more complex the environments, the more likely the mutation is not to be found.

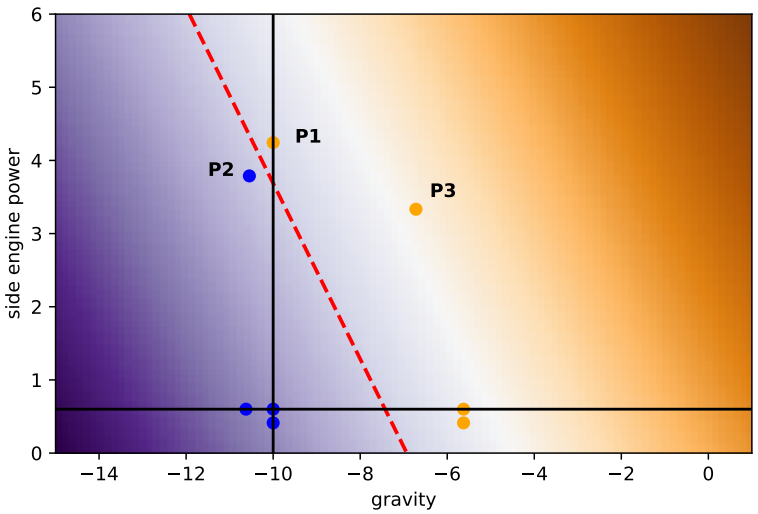


Figure 4.2 Generated test environments for *LunarLander/PPO/PAC_ReLU* and a potential way to separate them based on whether or not they detect the mutation. Orange points detect the mutation while the Blue ones do not. The origin is centred on the initial environments.

In the second step, it is possible to go beyond the raw number of test environments detecting a certain mutation and, instead, to inspect which test environment detects the mutation. Indeed, as we generated test environments by modifying the initial environment parameters, this can allow us to shed some light on which parameter a mutation might be more easily sensitive to. By doing so, we can determine which test environments can help identify certain potential faults, thus leading to some sort of fault-detection method. We will report one example using R to illustrate our point, and we refer the reader to our implementation repository [177] for all the raw results. In the case of *LunarLander/PPO/PAC_ReLU*, in Figure 4.2, test environments with lower (absolute) gravity compared to the initial environment detect the mutation while the ones with higher (absolute) gravity do not. Thus, the mutation seems to be affected somehow by the gravity parameter. When the gravity is close to the one in the initial environment, higher engine power will be crucial to detect the mutation.

Test environments are depicted as orange/blue dots, depending on whether or not they detect the mutation, and we can see they can be easily separated linearly following the previous description. In particular, we can see that both test environments $P1$ and $P2$ seem to be

close to some frontiers for this mutation since, while being close in parameters space, they lead to an opposite decision on the mutation. Note that the red frontier is arbitrary as we do not have access to the exact frontier, as it would be potentially too computationally expensive to find it, as we explained in Section 4.3.2. Thus, it could be possible to find environments between $P3$ and the initial environment that could still detect the mutation. We just know $P3$ is an environment for which the healthy agents' reward distribution is at the limit of being different from the distribution observed in the initial environment, but it gives no information on the distribution of the mutated agents. While it might not be possible to draw meaningful information for all mutations, especially on a reduced set of test environments, it shows nonetheless that generating test environments in that way also allows us to explore a potential link between parameters of the environments and their impact on the mutation.

RQa_{2-2} By mapping, for a given mutation, which generated test environments detect it or not, we can analyze which of the parameters of the test environments affect the decision to detect the mutation. This outlines some form of fault-localization method.

4.5.3 RQa_3. Properties of generated HOM

Following RQa_2, we gathered the non-trivial FOMs for each environment/algorithms/detection criterion (i.e., neither detected by all test environments nor detected by none), see Table 4.4. We need at least two FOMs to generate HOMs as we stick to HOMs of order 2. We then trained new mutated agents based on the gathered HOMs in the same way as FOMs. Finally, mutated agents were evaluated using the previously generated test environments depending on the mutation detection criterion (R or DtR), and the type of HOMs was analyzed based on the classification we introduced in Section 2.3.3. Results are presented in Table 4.5.

As we can see, following the procedure mentioned for RQa_2, we do not end up with many non-trivial FOMs. Hence, the pool of generated HOMs is relatively small, with some configurations not yielding any. Nonetheless, we managed to obtain 12 HOMs when the R mutation detection criterion was used and 23 with DtR but only in *LunarLander*. Among those, Non-Subsuming (NS) constitutes 50% of HOMs using R method but only 30% using DtR, the difference between the two methods potentially being explained by the increased sensitivity of DtR. The remaining Subsuming HOMs are mostly of the *Weakly Subsuming Coupled* (WSC) types and generally compose more than half the generated HOMs for each configuration, which is similar to results obtained by Jia et al [42] with HOM in some traditional software programs. Interestingly, we found 2 HOMs that fit the type *Strongly Subsuming Coupled*

Table 4.5 Types of HOMs generated using RQa_2 non-trivial FOMs. If no HOM was generated, then a "-" is used. ***NS***: Non Subsuming, ***WSC***: (Weakly) Subsuming Coupled, ***WSD***: (Weakly) Subsuming Decoupled, ***SSC***: Strongly Subsuming Coupled.

Environment	**DRL Algorithm**	**Detection Criteria**	**HOM types**				
			HOM	**NS**	**WSC**	**WSD**	**SSC**
CartPole	PPO	R	3	1	0	0	2
		DtR	-	-	-	-	-
	A2C	R	3	2	1	0	0
		DtR	-	-	-	-	-
	DQN	R	-	-	-	-	-
		DtR	-	-	-	-	-
LunarLander	PPO	R	1	1	0	0	0
		DtR	6	1	5	0	0
	A2C	R	5	2	3	0	0
		DtR	14	6	8	0	0
	DQN	R	-	-	-	-	-
		DtR	3	0	3	0	0

(SSC), that is, HOMs for which test environments detect said HOMs and also detect its constituent FOMs. Aside from those two, no other *SSC* and no *WSD* were found. Even in Jia et al. case, *SSC* represents a rare occurrence ($< 1\%$ of Subsuming HOMs) and so the fact we found none is not too surprising judging by our limited number of HOM/test environments. Nonetheless, we showed that Subsuming HOMs (even if *Weakly* ones), which are more complex and subtle mutants, could be generated and make up for more than half the HOM generated.

RQa_3 HOMs generated from non-trivial FOMs, while few are in the majority Subsuming HOMs, which de facto makes them more interesting cases to use as we pointed out in Section 2.3.3. If more subtle Subsuming HOMs such as *SSC* are not as widely represented, the fact that some can be generated shows that such property is reachable in DRL, too.

4.6 Threats To Validity

Construct validity: The design choices of MT could affect our results. We made sure that all the mutation detection definitions used or developed are based on existing approaches. Moreover, while the hyper-parameters used could play a role in declaring a mutation detected (*p-value*, threshold θ, etc.), we preferred to stick to the hyper-parameters given in each original implementation of the detection definitions and leave to future work the study of the influence of hyper-parameters. How we searched the parameters space to generate test environments

could also have impacted our results. Nonetheless, the goal was to get a simple and effective approach that could be automated and serve as a basic heuristic for future work. The rest of our methodology, such as the definition of the properties of HOMs, is grounded in the scientific literature and taken as initially defined.

Internal validity: Mutation operators chosen could affect our results. While we can not be exhaustive, we made sure to create operators based on existing faults or taxonomy and to provide sufficient diversity in terms of the effect on the code (*agent*-based, *environment*-based, *policy*-based). For the *environment-based* operators, while the probability of application of 1.0 might not be real to simulate sensor defaults, it was necessary to avoid potential bias over which step would be affected by the mutation when the probability is set to < 1.0. The environment parameters modified to generate the test environments could also impact the results. However, the goal was to show we could automatically generate simple and effective test environments that could be leveraged to analyze MT in DRL. As such, the choice of parameters on their own is not as relevant.

External validity: The choice of environment types/algorithms could impact the generalization of our experiments. We made sure to select two environments as well as three algorithms that are generally used as a benchmark in DRL.

Reliability validity: Implementing the mutations could lead to unintended faults. We used the Stable-Baselines framework as a basis to implement our mutations to lower the risk. We also make our implementation and artifacts available [177].

4.7 Chapter Summary

This Chapter presents a reliable framework for MT in DRL, *RLMutation*. We have defined three categories of FOMs based on real faults that can happen during the design and training of DRL agents, resulting in 11 mutation operators. We compared different mutation detection criteria choices based on the previous framework applying MT to DRL and DL. We also evaluated HOMs, a combination of FOMs that can prove to be more complex to detect, making for interesting faults to investigate. The FOMs used in the HOMs generation were selected based on their relevance to generated parameterized test environments. We have tested our approach on a set of state-of-the-art DRL algorithms (DQN, A2C, and PPO) over two benchmark environments (LunarLander and CartPole). The results of our study show that the mutation detection criterion choice is essential when it comes to detecting mutations (*ratio* vs. *distribution* of rewards). In particular, the new proposed criteria can detect respectively up to 25% / 45 % more mutations compared to the previous definition. We

demonstrate that by testing the agents in modified environments, we can detect non-trivial FOMs that are not detected by testing the agents in the environments in which they were trained. Finally, we show that 50% of the HOMs generated from non-trivial FOMs possess the Subsuming property, meaning that they are harder to detect than their constituent FOMs and more interesting from a testing perspective. Thus, we proposed an applicable framework to reliably find faults in DRL codes.

CHAPTER 5 A PROBABILISTIC FRAMEWORK FOR MUTATION TESTING IN DEEP NEURAL NETWORKS

The objective of this chapter is to redefine the statistical MT approach to account for the limitations of existing definitions that we partially showed in the previous chapter. We propose to rely on Bayesian probability to alleviate the inconsistency problem. Through evaluation using three models and eight mutation operators used in previously proposed MT methods, we show our definition leads to more coherent and stable results. For instance, we show that 25% of the mutations detected with the state-of-the-art MT definition should not be so when looking at the probabilistic analysis given by PMT. We also analyze the trade-off between the approximation error and the cost of our method in terms of model instances to train, showing that a relatively small error can be achieved for 100 instances.

5.1 Introduction

In the previous chapter, we saw how MT could be applied to DL, specifically DRL. We saw that stochasticity had to be acknowledged when applying MT in DL, as it can lead to contradictory results when using the same DNN. While we accounted for this stochasticity by using a distributional approach following Humbatova et al.'s DeepCrime [116], we already saw some limits of the statistical testing approach, which motivated the redefinition of the killing criteria from the Reward based to the Distance-to-Reward based. To extend what we already observed in the previous chapter, consider that when comparing n healthy (i.e., non-mutated) DNN instances, that is n DNNs trained with the same parameters, architecture and data but with a different initial seed, and n mutated DNN instances, the decision using DeepCrime's formulation (whether or not the test set detect the mutation) depends on the given set of DNN instances (both the n healthy and n mutated). This is to say, if we keep everything constant but change the instance sets, the decision may change when it should not. Worse, it can even be the case that, by chance, such an approach may declare a DNN mutated when compared against itself through the choice of healthy instances. This raises an interesting problem as we cannot recognize the entity identity. In a nutshell, we argue that current existing MT frameworks in the context of DNN, due to the inherent randomness of the paradigm, resemble a sort of flaky test [184], meaning that different mutation test results may be returned upon a new test run for the same test set.

This study proposes a framework, Probabilistic Mutation Testing (PMT), which adapts the current MT formulation for DNNs in the context of Bayesian estimation. We formulate the

current MT for the DNNs problem formally before developing a Bayesian method to frame the problem. We leverage the posterior distribution obtained to provide both summary analyses based on point estimates such as the Maximum A Posteriori and the notion of Credible Interval. We also devise simple criteria based on the posterior definition that can be used as a simple cut-out threshold for decision-making. We finally estimate the error of our framework based on the number of DNNs used. In particular, we show the relation between the definition of MT as it was defined and our current PMT approach in terms of trade-off.

The findings of this study are based on three models/datasets and eight mutation operators and show how the PMT framework, while more expensive than MT to apply, can alleviate the flakiness issue we mentioned previously, as PMT's results are more consistent and stable across mutations. In particular, we show among the set of presented mutations that 25% of those lead to MT detecting said mutation when PMT probabilistic analysis highlights the contrary. Moreover, the estimated error diminishes as we increase the number of DNNs used in the MT assessment, with a relatively small error starting at 100 DNNs and negligible towards 190 DNNs.

To structure the study in this chapter, we define two RQs:

RQb_1 *What are the benefits of the application of PMT compared to simple MT?* In this RQ, we aim to highlight the difference between simple MT and our proposed probabilistic version by exploring the difference between the two methods in terms of stability, consistency and fine-graininess of the analysis of the mutations, where our probabilistic approach should prove to improve on those aspects.

RQb_2 *What is the trade-off between the approximation error and the cost (sampling size and number of bootstrapped replications) of our method?* This RQ quantifies the error the different parts of our probabilistic approach can lead to, particularly regarding the trade-off with the number of DL model instances needed in the computation. A larger number of instances should lower the error at the expense of more model instances needed, thus resulting in a higher computation time.

5.2 Motivating Example

To understand the need for PMT, let us replicate the MT process as proposed in DeepCrime, as they provided a comprehensive replication package, and their tool constitutes one of the latest iterations of MT. We followed their approach on one model and mutation operator.

Remember that we exactly replicated the DeepCrime process as provided in their replication package. We chose the model using MNIST [55], with the same DeepCrime architecture and hyper-parameters, as well as the *delete_training_data* mutation operator (which removes a percentage of the data proportionally for each class). The choice of model/operator does not matter, as similar behaviour occurs for any model/operator we tested on (see Chapter 5.4.3).

First, we built and trained multiple sets of 200 DNN instances, that is, 200 DNNs with the same architecture, parameters and data but different initial seeds. A first set is the set of healthy instances (i.e., non-mutated) that is thus obtained; we then produce five different sets of instances applying the mutation *delete_training_data* with magnitudes ranging from 3.12 to 30.93, magnitudes being used by DeepCrime. At the end of the process, we obtain 1200 model instances. Finally, let us perform six experiments applying exactly the DeepCrime statistical test (see Equation 4.3.1 in Chapter 4) to assess whether a mutant is killed, using the same test set in all cases.

A description of the example can be found in Figure 5.1. In the first experiment, we divided "healthy" instances into two disjoint sets of 100. We pretended one of the two "healthy" sets contains "unknown" instances. We then randomly sampled k instances from the 100 "healthy" subset and k out of the "unknown" set and compared them, where $k = 20$, similar to DeepCrime's method. We repeated the sampling 100 times. We then averaged the number of times each "unknown" sample was declared "mutant" according to the statistical test used in DeepCrime, which gives us an estimation of the probability that a given "unknown" sample will be declared "mutant". When choosing the initial two partitions of "healthy" and "unknown", we repeat the entire process 50 times to avoid the potential sampling effect. For all other experiments, we did the same as above, sampling sets from the 100 "healthy" instances but this time contrasting them with sets obtained by sampling real "mutant" instances (separating experiments for each parameter magnitude). We applied the same procedure as in the first experiment for the rest.

Results of the procedure can be found in Table 5.1. Remember, if the mutation test is stable and not prone to the flakiness issue we described, we should have an average detection probability of 0 or 1 (within a small epsilon). That is the result of the mutation test is not reliant on the instances (both "healthy" and "mutated") used, and it either always returns that the mutation is detected or always that it is not.

If for the mutated instances with 30.93% of training data removed, we observe what was expected, this is not true for all other mutation magnitudes with the probability ranging from 0.13 to 0.85. Worse, the "healthy" instances, if we were to use them to see if the test would consider them as mutants, are considered as such in 6% of the mutation test

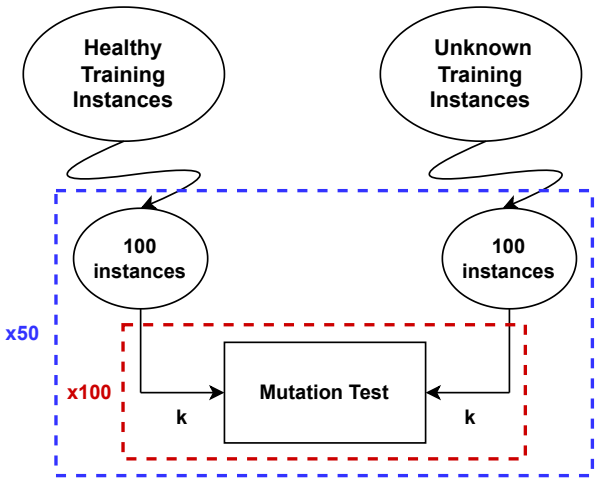


Figure 5.1 Replication of DeepCrime's mutation test with different instances. "Unknown" means either "Healthy" or "Mutant".

cases. Thus, it is clear that the current mutation test is not reliable in this form as it would imply different decisions depending on the instances one would use. If we were to put it into perspective, two users using the mutation test would end up with different results on a given mutation operator just because of the instances they trained for the test, even though the architecture of the model, the dataset, the learning process, and even the test set are the same, hence the flakiness we mentioned earlier.

Table 5.1 Average Probability of declaring "unknown" instances mutant after applying the experiments described in Figure 5.1 for both "healthy" ($\mathcal{I}$) and "mutated" (with "delete training data" mutation operator and different removal percentage) as "unknown" instances.

	Mutation parameters				
$\mathcal{I}$	3.12	9.29	12.38	18.57	30.93
0.06	0.13	0.45	0.47	0.85	1.00

Nonetheless, in all cases, we have some instances for which the mutation test result is that the mutation is detected. As such, if we follow the MT definition, the mutation is detected. However, one can see that this answer is not satisfactory given that, for instance, with the identity mutation, there is only a 6% probability on average that it happens. Thus, we argue that the question for MT, in the context of ML, **is not as much whether the mutation is detected or not, but rather *how* likely it is detected or not**. To put it into perspective, the idea is similar to traditional statistical tests with the concept of *p-value* and *effectsize*: having a significant *p-value* at a given threshold means there is some statistical difference, yet the difference can be so slight that it is not *practically* significant, which is

why *effectsize* is generally used to complement *p-value*. Thus, with MT, we showed there are likely some differences, yet the effect of the differences is not always the same.

5.3 Problem Definition

In this section, we will introduce the concepts and definitions relevant to the problem the chapter is tackling.

Definition 1 *For a DNN $\mathcal{N}$, let $\mathcal{D}$ be its training dataset, $\mathcal{A}$ its architecture (layers, hyper-parameters, ...) and $\mathcal{P}$ its learning process (optimizer, ...). Let $\mathcal{R}$ be the set of all possible pseudo-random number generators initialization (seeds), initial values, and random values (e.g., weights initialization, order of batch data, ...). We define an instance $f = (\mathcal{D}, \mathcal{A}, \mathcal{P}, r)$ of the DNN as the model obtained after initializing the DNN and performing the stochastic process of training it by using the initialization $r \in \mathcal{R}$.*

The set of all instances of the DNN achievable through the learning process $\mathcal{P}$ of architecture $\mathcal{A}$ over dataset $\mathcal{D}$ is:

$$\mathcal{F} = \{(\mathcal{D}, \mathcal{A}, \mathcal{P}, r) | r \in \mathcal{R}\}$$

The essential concept here is that r captures and models all the stochastic elements of the training process. For example, assuming all random values used in the training process (e.g., gradient descendant, weight initialization, and others) are derived from a pseudo-random number generator, just the initial random seed and the pseudo-random algorithm knowledge will suffice to ensure the deterministic replication of the entire process. Notice that there are infinite possible seeds and thus infinite possible concrete models (i.e., instantiations), each parameterized by a seed. If we use the object-oriented programming paradigm as a metaphor, a DNN $\mathcal{N}$ is a class whose attributes are of type $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{A}$; and any instance of it corresponds to an initialization r of the attributes (the weights of the layers, the order of data batch,...) followed by applying $\mathcal{P}$ on $\mathcal{N}$. Note that for practical purposes, despite $\mathcal{F}$ being infinite, it is represented with a finite number of bits. Thus, its realization (on a computer) contains a large but finite number of instances.

Definition 2 *Let $\mathcal{F}$ be the set defined in Definition 1 for a given DNN $\mathcal{N}$. Let $\mathcal{M}$ be a mutation of the DNN $\mathcal{N}$ induced either over $\mathcal{A}$, $\mathcal{P}$ or $\mathcal{D}$. We note the set of all instances achievable of the mutant $\mathcal{M}$ as:*

$$\mathcal{F}_{\mathcal{M}} = \{(\mathcal{M}(\mathcal{D}, \mathcal{A}, \mathcal{P}), r) | r \in \mathcal{R}\}$$

To simplify notation and for generality, we also consider the identity mutation $\mathcal{M} = I$ that is the mutation that doesn't alter the DNN, in which case $\mathcal{F}_I = \mathcal{F}$.

For instance, $\mathcal{M}$ can be "delete 3% of the training dataset". In that case, $\mathcal{M}$ is induced over $\mathcal{D}$. Note that if $\mathcal{F} \cap \mathcal{F}_\mathcal{M} = \emptyset$, it does not mean that elements of $\mathcal{F}$ and $\mathcal{F}_\mathcal{M}$ disagree on all possible values and for all possible instances. For example, it is possible that given $(f_1, f_2) \in \mathcal{F}$ and $f_\mathcal{M} \in \mathcal{F}_\mathcal{M}$, $\exists x$ input such that $f_1(x) = f_\mathcal{M}(x)$ and $f_2(x) \neq f_1(x)$ just as described in the introduction of this chapter. Yet, $f_\mathcal{M}$ is a mutated instance. In other words, on certain inputs, two "healthy" instances may disagree while they agree with a mutated instance. We can then define MT for DNN as follows:

Definition 3 *Let $\mathcal{F}$ and $\mathcal{F}_\mathcal{M}$ be the two sets of (non-empty and finite) instances as defined in Definition 1 and 2. Let $\#$ represent the cardinality of a set. Let $\mathcal{T}$ be a test set, and n_1, n_2 two positive integers (non-zeros, not necessarily equal). We define $S_\mathcal{F} = \{X \subset \mathcal{F} |\ \#X = n_1\}$ and $S_{\mathcal{F}_\mathcal{M}} = \{X \subset \mathcal{F}_\mathcal{M} |\ \#X = n_2\}$. MT for DNN is a function $Z_\mathcal{T}$ defined as:*

$$Z_\mathcal{T} : S_\mathcal{F} \times S_{\mathcal{F}_\mathcal{M}} \longrightarrow \{0, 1\}$$

In other words, given two sets of models' instances, the test decides if one is a mutated version of the other. Practically speaking, $Z_\mathcal{T}$ is a definition formulated to accommodate previously published MT functions. For example, testing one single healthy instance N_s against one (single) mutant instance N_m, (i.e., traditional MT), is captured in our definition by setting $S_\mathcal{F} = \{X \subset \mathcal{F} |\ \#X = 1\}$, $S_{\mathcal{F}_\mathcal{M}} = \{X \subset \mathcal{F}_\mathcal{M} |\ \#X = 1\}$ and $Z_\mathcal{T} : S_\mathcal{F} \times S_{\mathcal{F}_\mathcal{M}} \longrightarrow \delta_{N_s(\mathcal{T}), N_m(\mathcal{T})}$, where δ is the Kronecker delta.

Similarly, DeepCrime [116] mutation test is modeled by setting $S_\mathcal{F} = \{X \subset \mathcal{F} |\ \#X = n\}$, $S_{\mathcal{F}_\mathcal{M}} = \{X \subset \mathcal{F}_\mathcal{M} |\ \#X = n\}$ and $Z_\mathcal{T}$ (see Equation 4.3.1 in Chapter 4).

Note that PMT results depend on the test set $\mathcal{T}$ and on applied mutation operator $\mathcal{M}$, just like in traditional software engineering, but also on the sampled and compared DNN instances ("healthy" versus "mutated"). In practice, due to resources limitation, we don't have access to $\mathcal{F}$ and $\mathcal{F}_\mathcal{M}$ (and neither do we have $S_\mathcal{F}$ and $S_{\mathcal{F}_\mathcal{M}}$). Rather, we are working with $D_s \subset \mathcal{F}$ and $D_m \subset \mathcal{F}_\mathcal{M}$ representing the total number of "healthy" (respectively "mutated") trained and available instances. Thus $Z_\mathcal{T}$ turns out, in practice, to be $Z_{\mathcal{T}_{|S,S'}}$ where $S = \{X \subseteq D_s |\ \#X = n_1'\}$ and $S' = \{X \subseteq D_m |\ \#X = n_2'\}$. To avoid over-complicating the notations, we will refer to $Z_{\mathcal{T}_{|S,S'}}$ and $Z_\mathcal{T}$ as Z, since the objective is to have an approximation of a general function over $S_\mathcal{F}$ and $S_{\mathcal{F}_\mathcal{M}}$ and $\mathcal{T}$ is the same in all cases.

Ideally, this function Z should return 0 if $\mathcal{M} = I$ and 1 otherwise, or, at the very least, return

consistent results across any sets of instances for the same mutation operator $\mathcal{M}$. Yet, the example in Section 5.2 of this chapter showed it was not the case.

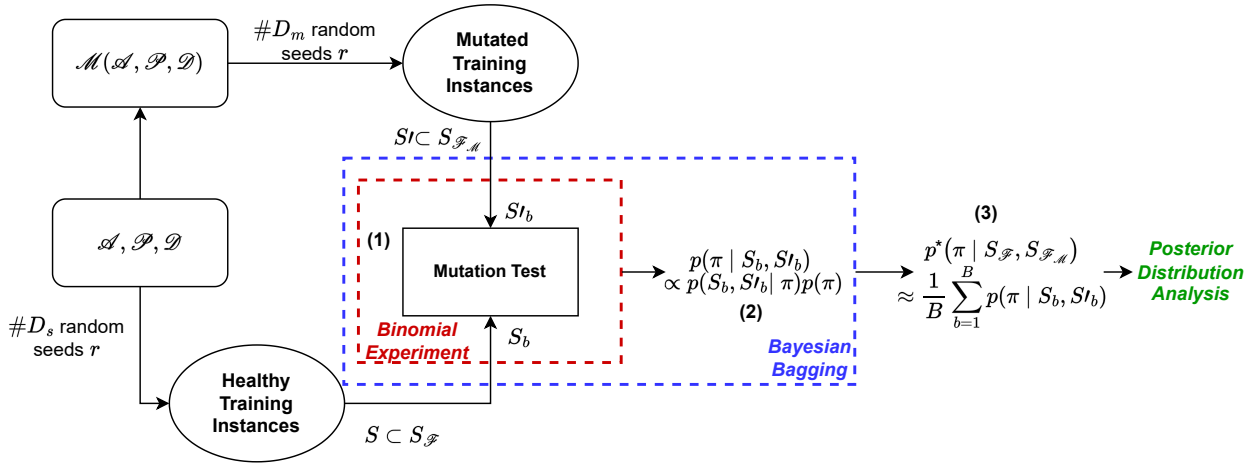


Figure 5.2 PMT methodology overview.

Having defined the setting we were working in and showing concretely the issue of current MT, we now describe our PMT framework. Figure 5.2 presents an overview describing the complete process.

5.3.1 A probabilistic framework for MT

Remember that MT compares some healthy instances of a DNN $\{N_1, ..., N_n\}$ against some mutated instances of a DNN $\{M_1, ..., M_n\}$, where n is a strictly positive integer, from a pool of instances that we defined in the previous section as S and S'. The key observation is that for any instance of $d_{s_i} \in S,\ d_{m_i} \in S'$ computing the decision function $Z(d_{s_i}, d_{m_i})$ correspond to performing a Bernoulli trial.

As such, instead of proposing a deterministic decision, it is possible instead to consider the probability of the outcome over $S,\ S'$. By repeating the comparison (i.e., experiment) N times, resampling at random each time, we define $Y = X_1 + ... + X_N \sim Binomial(N, \pi)$ where $X_i \sim Bernoulli(\pi)$ be the random variable representing the i^{th} realization of the mutation test Z (**(1)** in Fig 5.2).

Using Bayes rules, probability $p(\pi|S, S')$ estimation can be expressed as a Bayesian estimation problem for the parameter π knowing observed data S and S', that is:

$$p(\pi|S, S') \propto p(S, S'|\pi)p(\pi)$$

Since $p(S, S'|\pi)$ is Binomial, we can use the Beta distribution as conjugate prior. Since we have no information on the distribution, we can use a non-informative prior. Kerman [185] recommends using either the neutral prior (Beta($\frac{1}{3}$, $\frac{1}{3}$)) or the uniform prior (Beta(1,1)). In our experiment, we adopted the latter choice (i.e., Beta(1,1)). Overall, $p(\pi|S, S')$ distribution is $Beta(a + k, N - k + b)$ where $a = 1,\ b = 1$ are pseudo counts and k is the number of successes (see **(2)** in Fig 5.2).

5.3.2 Bayes Bagging

$p(\pi|S, S')$ and thus the distribution for parameter π are estimated over $S,\ S'$ rather than over $S_{\mathcal{F}},\ S_{\mathcal{F}_\mathcal{M}}$. Remember that $S_{\mathcal{F}},\ S_{\mathcal{F}_\mathcal{M}}$ cannot be accessed in practice; we are limited to finite subsets. A workaround to improve estimates is to exploit *Bayes Bag* [186], which consists of applying bagging to the Bayesian posterior.

Bagging (Bootstrap Aggregating) aims to generate B independent *bootstrapped* datasets by sampling with replacement from a given dataset D, following the established *bootstrap* methodology [187]. Each bootstrapped dataset is used with a predictor, and the B predictors' results are then aggregated (for instance, by averaging their predictions), which generally improves accuracy [188]. It is one of the techniques of the *ensemble learning* methods family [189]. *Bayes Bag* [186] applies this principle with Bayesian posteriors instead of predictors by obtaining B bootstrapped posteriors and averaging them to obtain a bagged posterior. In particular, Huggins [190] showed that Bayes Bag can result in more accurate uncertainty even with a limited number of replications ($B = 50$ or 100).

In the context of PMT, using Bayes bagging, we can obtain multiple bootstrapped posterior $p(\pi|S_b, S'_b)$ which can be then aggregated into $\frac{1}{B}\sum_{b=1}^{B} p(\pi|S_b, S'_b)$. This allows us to obtain the $p^*(\pi|S_{\mathcal{F}}, S_{\mathcal{F}_\mathcal{M}})$ (**(3)** in Fig 5.2) approximation as we wanted.

5.3.3 PMT posterior analysis

Once we plug the approximation of the posterior probability $p^*(\pi|S_{\mathcal{F}}, S_{\mathcal{F}_\mathcal{M}})$ into the PMT framework, Definition 3 is extended as:

Definition 3.bis *For $S_{\mathcal{F}}$ and $S_{\mathcal{F}_\mathcal{M}}$, $\mathcal{T}$ a test set, and Z a mutation function as defined in Definition 3. We define a probabilistic MT function ProbZ as:*

$$ProbZ_{\mathcal{T},Z,N} : (S_{\mathcal{F}} \times S_{\mathcal{F}_\mathcal{M}})^N \longrightarrow \pi|S_{\mathcal{F}}, S_{\mathcal{F}_\mathcal{M}}$$

Definition 3.bis provides a means to analyze the behaviour of the test set against the mu-

tations through the analysis of the obtained posterior. Posterior can be analyzed leveraging estimates widely used in Bayesian settings, for instance:

Point estimate: One can derive a point estimate relying on the posterior distribution. For instance, the Maximum A Posteriori (MAP), i.e., $\pi^{MAP} = argmax_{\pi} p^*(\pi|S_{\mathcal{F}}, S_{\mathcal{F}_{\mathcal{M}}})$ or the Minimum Mean Square Error (MMSE) $\hat{\pi} = \mathbb{E}(\pi|S_{\mathcal{F}}, S_{\mathcal{F}_{\mathcal{M}}})$.

Credible Interval: A point estimate is complemented using a *Credible Interval* (CI). This is the interval within which an unobserved parameter value is present with a given probability $1 - \epsilon\%$, that is, $p(\pi \in CI) = 1 - \epsilon$. Notice that, CI differs from a *confidence interval* [191]. Multiple CI exists, which can be tailored based on the point estimate used, such as the *Equal-tailed* CI for the median estimator, the *Highest density interval* CI for the mode estimator or the CI centred around the mean. For instance, the chosen CI can be used to measure the uncertainty of the previous probability (the wider the CI, the more uncertain the beliefs).

5.3.4 Effect analysis

If posterior analysis can shed some light on the behaviour of the mutations, we propose a practical criterion to establish if a mutation is *likely* killed.

Remember, in the ideal case, we would like our mutation test to return either always *not-mutant* or always *mutant* for any instance used. The motivating example of Section 5.2 showed it was not the case. However, the two resulting posteriors we could derive from those ideal cases can be leveraged as comparison points to calculate some similarity concerning the bagged posterior obtained for a given mutation. One way to compute such probability similarity involves using the Hellinger distance [176], which can be defined for two beta distributions $P \sim Beta(\alpha_1, \alpha_2),\ Q \sim Beta(\alpha_2, \beta_2)$ as:

$$H(P, Q) = \sqrt{1 - \frac{B(\frac{\alpha_1+\alpha_2}{2}, \frac{\beta_1+\beta_2}{2})}{\sqrt{B(\alpha_1, \beta_1)B(\alpha_2, \beta_2)}}}$$

where B is the beta function.

We have $0 \leq H(P, Q) \leq 1$, with a distance $H(P, Q)$ of 0, implying that both P, Q are the same. Thus, it's possible to calculate the distance between the bagged posteriors of a given mutation and both ideal posteriors we mentioned earlier. Then, we can compute a ratio of similarity between the two distances:

$$\mathcal{R} = \frac{H(P, Q_H)}{H(P, Q_M)}$$

where P is the bagged posterior, Q_H the ideal posterior with all *non-mutant* results and Q_M the ideal posterior with all *mutant* results.

A ratio $\mathcal{R}$ of 1 means that the bagged posterior is as similar to both ideal posteriors. In that case, we have little information on the practical effect of the mutation being detected, which can, for instance, happen for a posterior centred around 0.5 (i.e., 50% chance on average that MT returns *mutant* as a result for any instance set). A ratio higher than 1 implies the posterior is more similar to the ideal mutant posterior, and the opposite if the ratio is lower than 1.

To decide the magnitude of the effect, we elaborated the following empirical scale based on our results, inspired by existing empirical scale elaborated for *effect size* criteria such as Cohen's d [192], where $|d| > 1.20$ is *very large*, $|d| > 0.8$ is *large*, $|d| > 0.5$ is *medium* and $|d| > 0.2$ is *small*. In our case, we found empirically the ratio of similarity calculated with the healthy posterior to be around 0.82 at most. Thus, since we know the healthy instances are not mutation and should not be considered as such, any ratio below 0.82 illustrates a *very strong* evidence *against* the mutation being killed. From there, we can build the scale using the above-mentioned rule of thumb, mirroring our ratio, with 0.82 in our case being 1.20 for Cohen's d and 1 in our case being their 0. For the case above 1, we take the invert of the boundaries we would get in the case below 1. This leads to: $0.82 - 0.87$ (resp. $1.15 - 1.22$) *strong*, $0.87 - 0.92$ (resp. $1.09 - 1.15$) *medium*, $0.92 - 0.97$ (resp. $1.03 - 1.09$) *weak*, $0.97 - 1.03$ *negligible*. The decision is left to the user when considering a mutation *likely* detected based on the posterior or similarity ratio obtained, using some thresholds. Note that, in that configuration, we have three potential outcomes: the mutation is *likely* detected, the mutation is *likely* not detected, and no evidence points in either direction, which can happen when the thresholds are not met in either way (killed or not killed), that is we do not have enough evidence to point in either direction. In practice, this choice can default to not detecting the mutation.

With what was said before, we need to redefine the traditional mutation score. The mutation score is generally defined as:

$$MS = \frac{\#\text{mutations detected}}{\#\text{mutations}} \tag{5.1}$$

that is, the number of mutations detected over the total number of mutations. With our approach, we extend the mutation score to:

$$MS = \frac{\#\text{mutations} \mid \mathcal{R} > \theta}{\#\text{mutations}} \tag{5.2}$$

that is the number of mutations for which the similarity ratio is above a certain threshold θ (i.e., *likely* detected) over the total number of mutations considered.

An example that leverages the complete methodology for the decision will be presented in Section 5.4.3.

5.3.5 Error estimation

There are two types of error we aim to quantify while using PMT: the error of the bagging process (i.e., if the choice of the bootstrapped data influences the bagged posterior results) and the error of the sample representativity (i.e., given a specific size, does the choice of the sampled population of instances affect the bagged posterior results). Section 5.4.3 will analyze the two errors.

Once the bagged posterior $p^*(\pi|S_{\mathcal{F}}, S_{\mathcal{F}_{\mathcal{M}}})$ is available, one can obtain error approximation. This is formulated to estimate a Monte-Carlo Error (MCE) [190]. MCE estimation has been widely studied [193].
We evaluate the MCE as follows: consider R replications of our bagged posterior as Monte-Carlo simulations from which we will derive the values of Definition 3.bis. One can estimate the MCE using, for instance, jackknife bootstrapping [194]. In other words, we consider our R replications of the bagged posterior $\{\pi_1|S_{\mathcal{F}}, S_{\mathcal{F}_{\mathcal{M}}}, \ldots, \pi_R|S_{\mathcal{F}}, S_{\mathcal{F}_{\mathcal{M}}}\}$ from which we can extract a desired estimate e_i such as $X = \{e_1, \ldots, e_R\}$.

We used the jackknife formula described in [193] and complemented the error estimation with a traditional confidence interval over the estimate recommended by Koehler and Brown [193].

We will track the estimates of the mean μ and variance var of the bagged posteriors obtained. A straightforward Monte-Carlo estimator for both is simply the average over each replicate's μ_i and var_i.

5.4 Experimental Setup

The goal of this evaluation is to investigate the following research questions:

RQb_1 What are the benefits of the application of PMT compared to simple MT?

RQb_2 What is the trade-off between the approximation error and the cost (sampling size and number of bootstrapped replications) of our method?

5.4.1 Datasets and Mutations

In our evaluations, we use three models/datasets previously leveraged by DeepCrime to show how PMT alleviates the problem of flakiness mentioned earlier and to compare it to the latest designed MT, i.e., DeepCrime's definition of MT (see Equation 4.3.1). More precisely, the following model/dataset combinations: MNIST [55] (MN) along with an 8-layered convolutional neural network [195]; MovieLens dataset [196] to train the Movie Recommender [197] (MR) model; A synthesized UnityEyes (UE) dataset [198] along with a specific model [199].

Table 5.2 Systems under test. For each model, we provide the average metric value and the standard deviation (in parenthesis).

ID	Training Data	Test Data	Epochs	Metric	Value
MN	60,000	10,000	12	Accuracy	99.15 (0.06)
MR	72,601	18,151	12	MSE	0.047 (0.001)
UE	103,428	25,857	50	Angle based	2.6° (0.2)

Table 5.2 shows the average metric values obtained on the test set across all our "healthy" DNN trained instances; figures are in line with DeepCrime reported values. As UE and MR systems are regression-based, we considered (as in DeepCrime) that a prediction is accurate if it differs from the correct one by no more than one rating (for MR) or if the angle is no more than 5 degrees (for UE).

Regarding mutations, we chose both *source-level* and *model-level* mutations.

Source-level mutations are extracted from the detailed mutation operators proposed in DeepCrime. To select which mutation to investigate in priority (and limit the number of instances to train), we used DeepCrime Killability and Triviality metrics. In their paper, *Killability* is defined as whether or not the training data detects a mutation operator configuration using the statistical test. *Triviality* roughly quantifies how easily any test input of the test set can detect a mutation operator. To push the method to its limits, for each dataset/model, we selected mutations with Killability/Triviality that is the highest/lowest possible. Finally, we also chose some common mutations for all models to have common points of comparison for all models. Since mutations' Killability/Triviality is for a *given model* in DeepCrime, interesting mutation operators concerning those criteria in one case will not necessarily be interesting for another. That is why, in Table 5.3, some mutation operators are not applied to some models/datasets.

Regarding *Model-level* mutations, DeepCrime proposes some of those mutations but does not analyze or implement them. Therefore, we chose instead to leverage some mutation operators proposed and analyzed in MuNN [114] and DeepMutation [112]. Note that those mutations

Table 5.3 Mutations (Source / Model Level) chosen for each dataset/model. '✓' means selected, and '-' means not selected.

	TCL	TRD	WCI	ACH	TUD	LCH	OCH	AWF	FNO
MN	✓	✓	✓	✓	-	-	-	✓	✓
MR	✓	✓	-	-	✓	✓	-	-	-
UE	✓	✓	-	-	-	✓	✓	✓	✓

do not apply to *MR* as the implementation differs from a traditional neural network structure and would require modifying the operators to fit.

Table 5.3 provides an overall view of the selected mutations (✓) for a given model/dataset, with the mutation acronym being described below:

- *change_label (TCL)*: Modify a percentage of training data labels, replacing them with the most frequent label in the dataset.
- *delete_training_data (TRD)*: Remove a portion of the training dataset from each class proportionally.
- *change_weights_initialisation (WCI)*: Change how weights are initialized in all the model layers.
- *change_activation_function (ACH)*: Change the (non-linear) activation function of a layer by another (non-linear) activation function.
- *unbalance_training_data (TUD)*: Remove a portion of data belonging to the classes whose apparition frequency is less than average.
- *change_loss_function (LCH)*: Change the loss function to another loss function.
- *change_optimisation_function (OCH)*: Change the optimisation function to another optimisation function.
- *add_weights_fuzzing (AWF)*: Add gaussian noise of magnitude σ to a certain percentage of weights of a layer.
- *freeze_neurons_output (FNO)*: Freeze (delete) a percentage of neurons of a given layer.

DeepCrime's mutations were reused precisely as provided in the replication package. MuNN [114]/DeepMutation [112] based mutations (the two last ones) were implemented based on the description/parameters provided in the papers.

5.4.2 Instrumentation and parameters

To carry out the experiments, we use the same requirements as documented in DeepCrime [116], namely, Python (3.8), Keras (2.4.3), and Tensorflow (2.3). We also used the models/-datasets, mutation operators as well as the MT procedure used in their replication package. For each mutated/healthy model, we train 200 instances and then evaluate the accuracy of each instance on the dataset test set.

Unless specified otherwise, all experiments use the following default parameters: $N = 100$, the number of trials for each Binomial experiment, and $B = 100$, the number of bootstrap repetitions. Again, such values are a compromise to ensure a trade-off between having a sufficient number of evaluations and keeping the computation within a manageable time. Moreover, we used the same number of instances as in DeepCrime ($n = 20$) for the MT with the same MT function Z that they used (see Equation 4.3.1).

First Experiment

The first experiment aims to apply PMT to the previously listed models/mutations and to draw a comparison with MT. We leveraged 200 training instances per model and dataset and mutation for our method to do this. As a point of comparison, we will apply MT on DeepCrime's instances provided in their replication package [116]. We implemented and applied the procedure detailed in Section 5.3.1. The procedure was needed to obtain a bagged posterior for each mutation (including the identity mutation, that is, the "healthy" instances) of each dataset/model. From there, we can leverage the effect analysis method we introduced in Section 5.3.3 to calculate the ratio of similarity obtained for each mutation and compare it to the results one would obtain with simple MT to nuance them.

Second Experiment

The second experiment aims to evaluate the error over the bagged posterior approximation and the sampled population representativity.

To estimate the bagged posterior approximation, we repeated $N_{exp} = 100$ times the calculation of the bagged posterior approximation, using the jackknife estimation as explained in Section 5.3.5.

To evaluate representativity, we considered the following. Since the trained instances are "sampled" at random when trained (i.e., the random seeds used in training are equally likely to be picked), the representativity of the sampled instances will depend on their number.

Thus, we repeated the MCE estimation to estimate the bagged posterior approximation with a different number of sample instances (from 25 to 190). We repeated this process $N_{pop} = 30$ times to account for the possible effect of the choice of the samples over the obtained bagged posterior. In other words, from the 200 training instances, we repeated the jackknife estimation 30 times, each time with a different sampled population of the same size. This allowed us to examine the evolution of the average parameters estimate μ and var across the bagged posterior and the average of their approximation error boundaries based on the sample size and the sampled instances. As described in Section 5.3.5, we also compute confidence interval values as recommended in the literature [193].

5.4.3 Results

In this section, we present a representative sample of our results, and we provide all the results in our replication package [200].

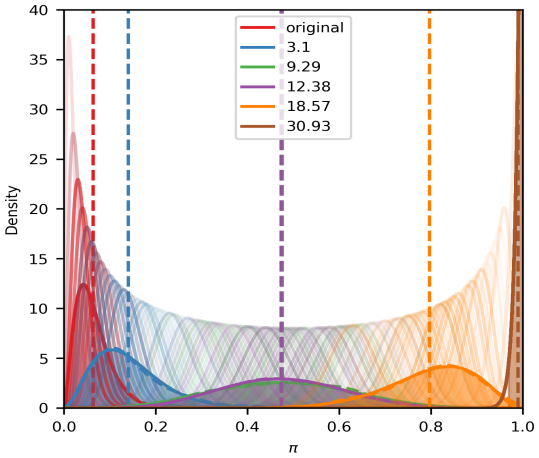
RQb_1: PMT benefits over simple MT

We report results for two mutation operators for each model, and the rest can be found in the replication package [200]. We first report the obtained posteriors in Figure 5.3. Each curve represents the posterior distribution for a given mutation operator magnitude following the procedure described in Section 5.3.1. For instance, in Figure 5.3a, the plain orange curve represents the bagged posterior distribution of the probability of killing the *delete_training_data* mutation with magnitude 18.57. Its MMSE point estimate is $\hat{\pi} = 0.8$ (vertical dash line), and the credible interval width is $|CI| = 0.4$ (coloured area). Transparent lines are the bootstrapped posteriors obtained from each bootstrapped data S_b, S'_b (see Section 5.3.2). We then report in Table 5.4 a comparison between simple MT results for each of the mutation operators (i.e., 1 for the mutation is detected, 0 if it's not) and the ratio of similarity (with the effect) as we defined in Section 5.3.3 for PMT. For instance, for the mutation $MN - TRD$, the magnitude 9.29 was considered detected by MT. Yet, we found a *negligible* effect when using our ratio metric, i.e., there is no strong argument to point out that the mutation is either *likely* detected or *likely* not detected. By default, the user can consider it not to be detected to avoid potential false positives (i.e., considering a mutation detected when it is not). These results will allow us to showcase the advantage of PMT over MT. Overall, among the mutation considered, we observe that 25% of the mutations that MT detected turned out not to have an informative posterior according to PMT and so should not be considered detected.

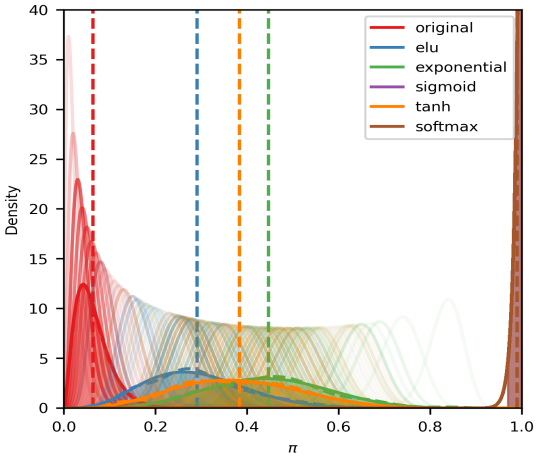
Stability: One thing we first showed with the motivating example and that we show again

Figure 5.3 Posterior distribution for mutation operators of different magnitudes. Vertical dash lines symbolize the point estimate value of each posterior. Plain lines curve represent the bagged posteriors, with the coloured area underneath being the *CI*, while transparent lines are each posterior obtained from bootstrapped data.

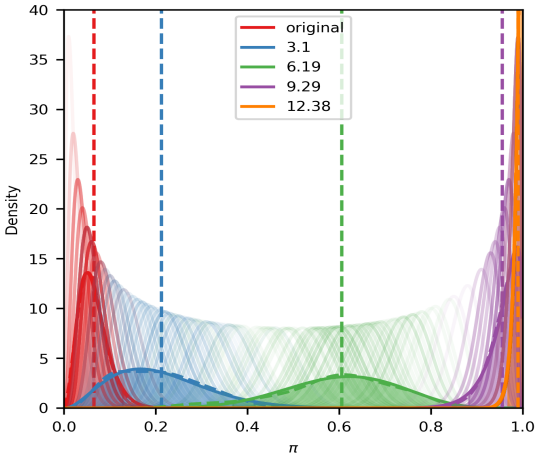
(a) MN - TRD



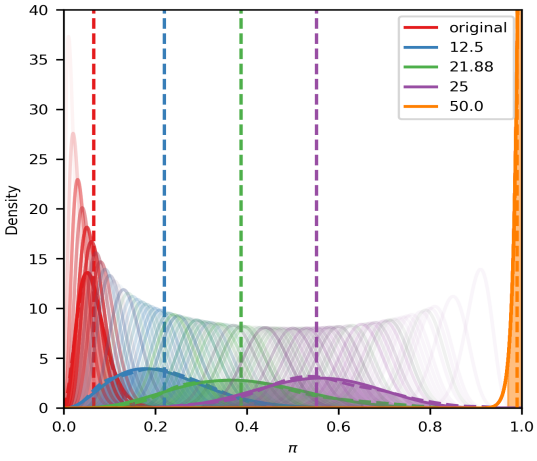
(d) MN - ACH



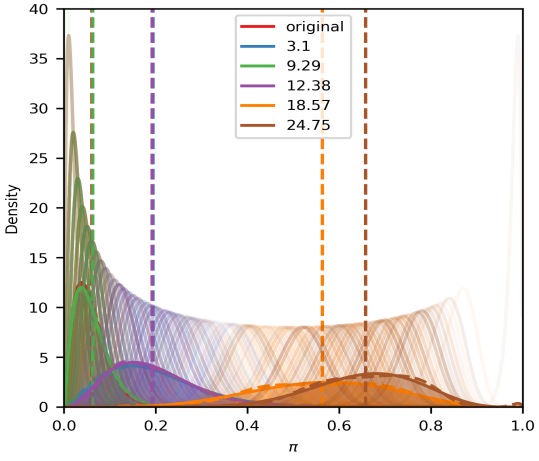
(b) MR - TRD



(e) MR - TUD



(c) UE - TRD



(f) UE - TCL

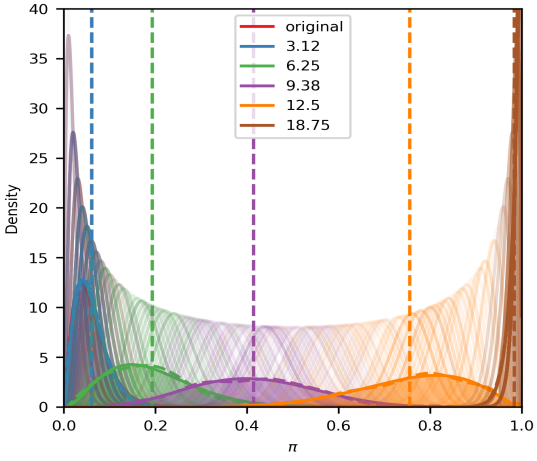


Table 5.4 Example. For MT, "✓" means the mutation is detected and "✗" means the mutation is not detected. For PMT, we give the ratio of similarity value and the effect based on the scale presented in Section 5.3.3 with the following meaning: "∘" *negligible*, "-" *weak*, "±" *medium*, "+" *strong* and "++" *very strong*. red is when the effect is the likeliness of the mutation being detected ($\mathcal{R} > 1$), and blue when the effect is the likeness of the mutation not being detected ($\mathcal{R} < 1$). We also present the similarity ratio when comparing healthy instances against themselves (identity mutation $\mathcal{I}$).

$\mathcal{I}$	MN - TRD					MN - ACL				
0.76	3.1	9.29	12.38	18.57	30.93	elu	exp	sigmoid	tanh	softmax
MT	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
PMT	0.91 (±)	1.00 (∘)	1.00 (∘)	1.05 (-)	>2 (++)	0.99 (∘)	1.00 (∘)	>2 (++)	1.00 (∘)	>2 (++)
$\mathcal{I}$	MR - TRD					MR - TUD				
0.81	3.1	6.19	9.29	12.38		12.5	21.88	25	50.0	
MT	✗	✓	✓	✓		✗	✗	✗	✓	
PMT	0.95 (-)	1.00 (∘)	1.86 (++)	>2 (++)		0.96 (-)	1.00 (∘)	1.00 (∘)	>2 (++)	
$\mathcal{I}$	UE - TRD					UE - TCL				
0.73	3.1	9.29	12.38	18.57	24.75	3.12	6.25	9.38	12.5	18.75
MT	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓
PMT	0.94 (-)	0.72 (++)	0.95 (-)	1.00 (∘)	1.00 (∘)	0.74 (++)	0.95 (-)	1.00 (∘)	1.05 (-)	>2 (++)

here is the lack of stability of the simple MT, i.e., the flakiness we mentioned earlier. Indeed, the fact that all posterior distributions do not translate to the ideal mutant or not-mutant posterior we described (that is, MT always returning 0 or always 1 no matter the instances used) can have dire consequences. For instance, looking at Figure 5.3c, PMT shows that the posterior distribution of the mutation of magnitude 9.29 is very similar to the healthy one. More directly, in Table 5.4, we see that the mutation is *likely* not detected with a similarity ratio of **0.72** (very strong). Nonetheless, the point estimate is non-zeros, which means that, for some instances, there is a chance that simple MT returns "mutant" as a result, despite PMT showing strong evidence the mutation should not be considered detected. This is similar to our motivating example in Section 5.2 where we, for instance, found out that 6% of tests done on healthy instances returned mutant as a result despite no mutation being present.

Finding 1: PMT allows stability over test results contrary to MT. That is, the decision made over a given mutation is taken while accounting for any instance possible, which prevents the flakiness issue we illustrated previously, i.e., MT returning 0 and 1 for the same mutation depending on the instances used in the test.

Consistence: Besides mitigating stability problems, PMT allows tackling another issue of MT: the potential lack of consistency across the tests. Indeed, as MT outcome is binary, one can not ensure that mutations that behave similarly lead to the same MT outcome since

there is no information available for the posterior of the distribution. On the contrary, using PMT, one can compare results for different mutations, whether from the same operators or a different one. For instance, in Figure 5.3a both the mutation of magnitude 9.29 and 12.38 exhibit the same posterior and a similar ratio of similarity of **1.00** (negligible), as such, logically, a decision made over these two mutations should be the same. However, using Deepcrime's instances for MT yielded opposite results, probably because of the training instances used in the test.

Finding 2: PMT allows for coherence across results, ensuring the test outcome will be similar for mutations that exhibit similar posterior distributions/similarity ratios contrary to MT.

Granularity: Finally, note that using PMT, one can quantify if a mutation operator is more or less *likely* to be detected or not detected. For instance, in Figure 5.3f, for UnityEyes (UE) $TCL - 3.12$, the posterior distribution is similar to the one of the original (i.e., healthy) model and exhibits a similarity ratio of **0.74**. Thus, it will not be detected by the test set for many sets of instances, so there is strong proof for considering the mutation *likely* not detected. This analysis can not be inferred from simple MT, as it returns a deterministic decision over the given instances. Similarly, for $TCL - 6.25$, we only have a weak effect to consider the mutation *likely* not detected, which is still more than for $TCL - 9.38$ where the effect is negligible with a ratio of **1.00**. As such, if there are some incentives to say that indeed $TCL - 6.25$ can be considered *likely* not detected (and thus there is evidence that it should not be considered detected), such incentives do not exist for $TCL - 9.38$. Nonetheless, MT would consider those two mutations to be similarly "not detected", which limits the potential to analyze them. Similarly, between $TCL - 12.5$ and $TCL - 18.75$, which are both considered detected by MT, our approach highlights that we have more incentives to consider the mutation *likely* detected for the latter rather than for the former.

Finding 3: PMT delivers a finer-grain analysis of the mutations, which allows us to compare them on the *likeliness* of the mutation being detected. This might enable, for instance, to adjust potential thresholds one user would select to consider a mutation *likely* detected, depending on the similarity ratio obtained. This is impossible with MT, which will consider all the detected mutations similarly.

RQb_2: PMT trade-off study

Figure 5.4 and 5.5 show the error estimation of the estimates when computing the bagged posterior with different samples and sample sizes. Figure 5.4 focuses on one model/mutation operator and varies the magnitude of the mutation. In contrast, Figure 5.5 shows the results for the same mutation operator and the same magnitude for different models. Similar trends can be noticed for other models/mutation operators. First, from these graphs, we can make the following general observations:

- The larger the sample size, the lower the error across the samples of the same size. This resonates with the intuition that the bootstrap hypothesis is increasingly valid. In other words, the sample is increasingly more representative of the unknown underlying population as we increase the number of instances in the sample.
- The average across the $N_{pop} = 30$ runs of the different estimates, as well as the average of their lower and upper bound (dot on the plots), are close. This suggests that the individual confidence interval, on average, is not large for a given sample. In a nutshell, this means there is no huge variation between the bagged posterior obtained from the Monte-Carlo simulation for a given sample. Overall, our findings suggest a low Monte-Carlo error when estimating the bagged posterior approximation error for $B = 100$ bootstrapped datasets, similar to Huggins [190] observations.

As a consequence of these findings, if the bagged posterior approximation error is low, the error due to the representativity (and so the size) of the sampled population is big. Yet, it will decrease logically as the sample size increases. Of course, the larger the number of available trained instances, the better. In practice, our choice of 200 instances seems warranted, as the variation across samples decreases with the sample size, and for 190 the confidence intervals are relatively small.

Finding 4: When applying PMT, following Huggins [190] observation of setting $B = 100$ for the bootstrapped repetitions is a sound choice. Moreover, leveraging a sampled population of 200 also seems warranted as the approximation error over the bagged posterior is relatively small with a sample size of 190.

Secondly, we can now compare the evolution of the error estimation across models and mutations. In Figure 5.4, we can compare the evolution through the increased magnitude of the mutations. We note, for instance, that the error estimation tends to be lower for a mutation operator with a low or high magnitude compared to a medium one (3.1 and

30.93 vs 9.29, *glorot normal* and *zeros* vs *he normal*). Most likely, mutations with medium magnitude (for a given mutation operator) are more prone to divergence among the instances and more likely to have larger differences across samples. In Figure 5.5, we compare the error estimation for the same mutation operator and magnitude across the models. There does not seem to necessarily be a similar evolution across models for the same mutation operator (see, for instance, "change_label", with *UnityEyes* and the others). As such, the error estimation is model-dependent and does not seem to be based on the mutation operator.

Finding 5: Medium magnitude mutation operators tend to have a higher error for the same sampled size when compared to mutation operators with low/high magnitude. Moreover, there is no explicit trend in the error decrease for the same mutation operator across the different models, so the error reduction seems more model-dependent.

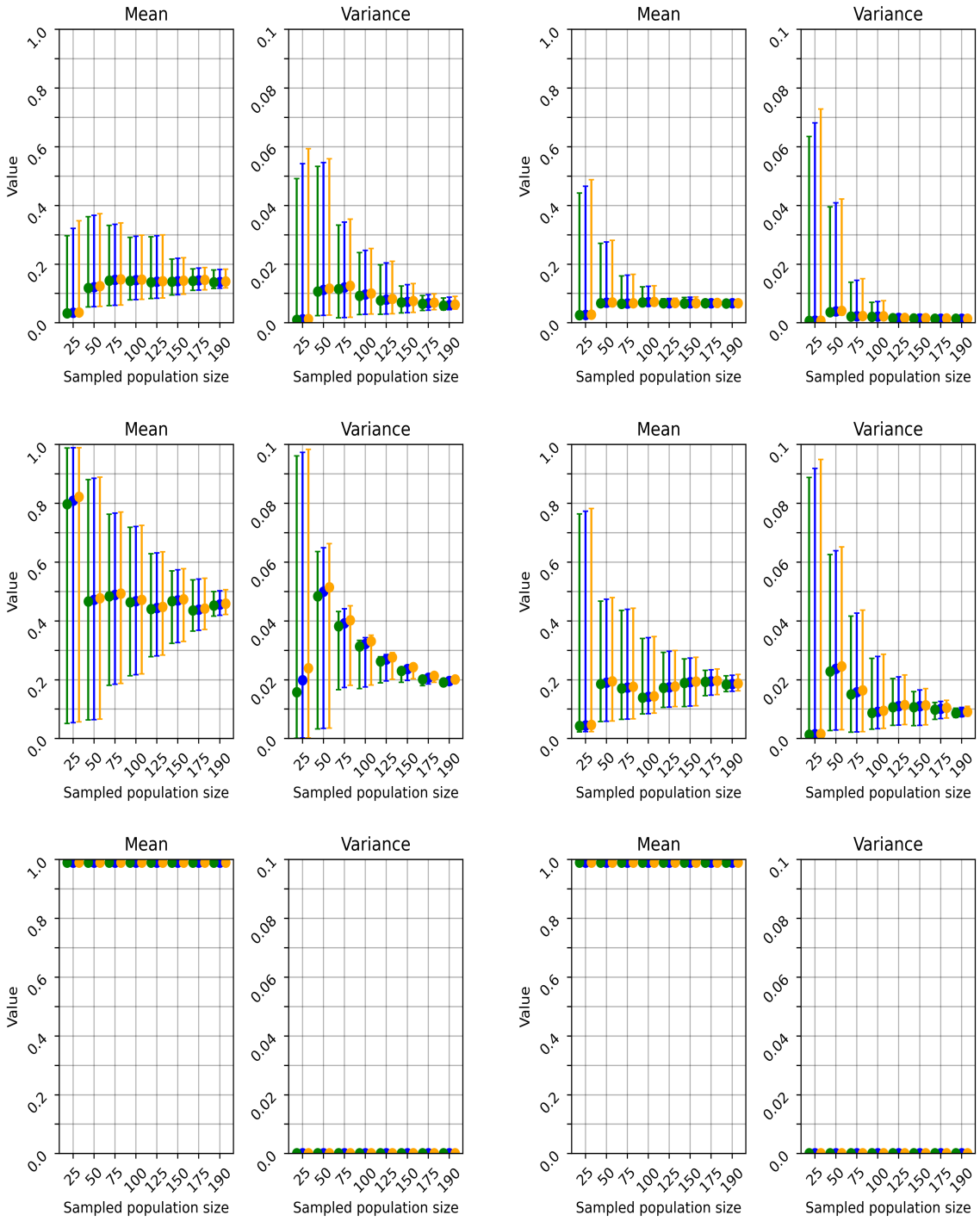


Figure 5.4 Error of estimates when testing on "delete_training_data" (left) and "change_weights_initialisation" (right) mutation operators for MNIST on three magnitudes (top to bottom: 3.1, 9.29, 30.93 and *glorot normal*, *he normal* and *zeros*). For each estimate, we present the average over $N_{pop} = 30$ of the monte-carlo estimate (blue), monte-carlo lower bound (green) and monte-carlo upper bound (orange) as calculated in Section 5.3.5. We also display the 95% confidence interval.

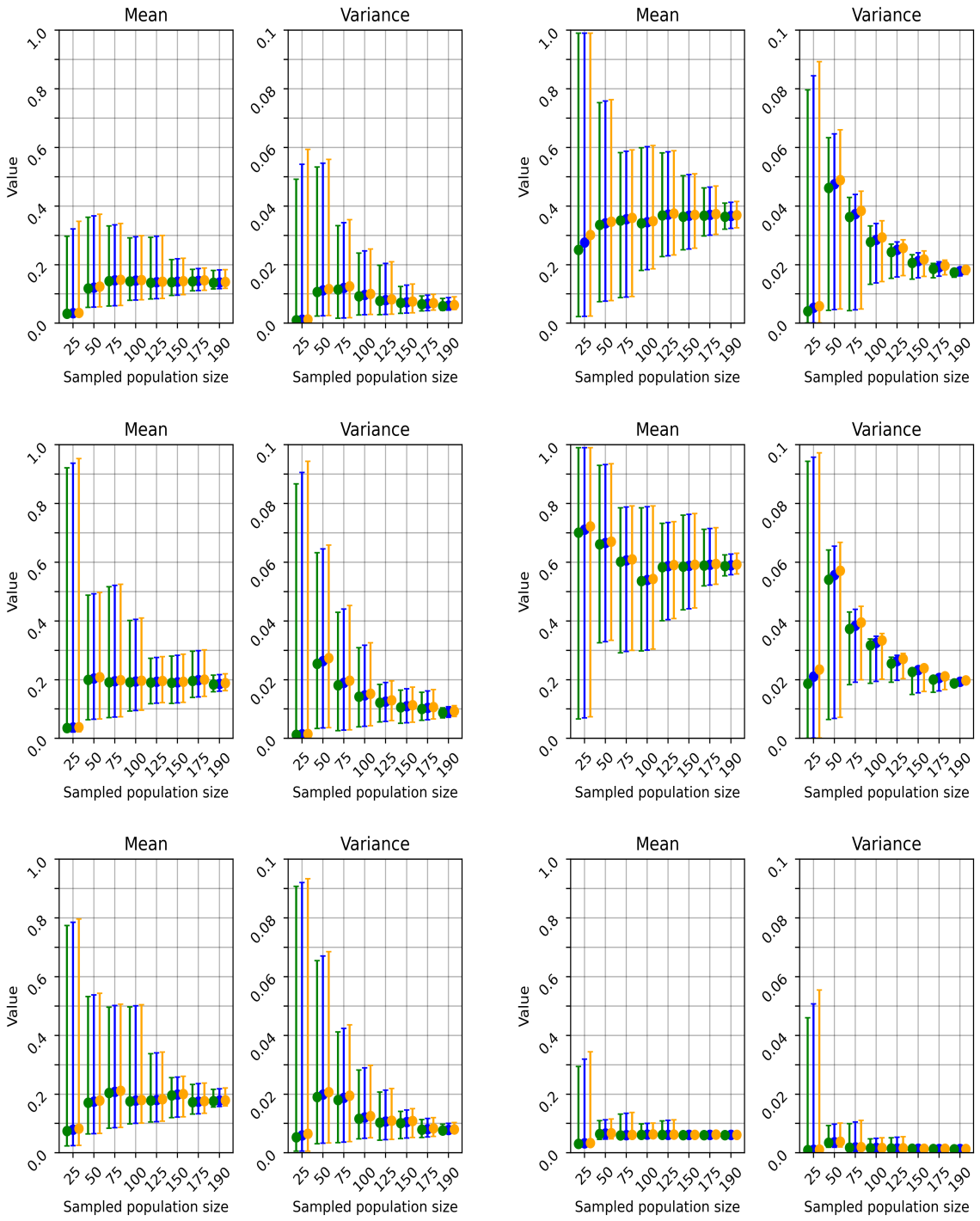


Figure 5.5 Error of estimates when testing on "delete_training_data" (left) and "change_label" (right) mutation operators for the three models the same magnitude (top to bottom: $MNIST$, $MovieRecomm$, $UnityEyes$). For each estimate, we present the average over $N_{pop} = 30$ of the monte-carlo estimate (blue), monte-carlo lower bound (green) and monte-carlo upper bound (orange) as calculated in Section 5.3.5. We also display the 95% confidence interval.

5.5 Discussion

In light of our results, we briefly discuss a few important points.

5.5.1 Generalization of the results

As we mentioned at the beginning of Section 5.4.3, we only presented a limited number of results because of space limitations and put the rest in the replication package [200]. We briefly discuss the rest of the results and invite the reader to check the replication package.

Regarding RQb_1, the results presented are the ones that better illustrate the usefulness of the methods, with a wide range of diverse posteriors among the magnitude of the same mutations, as well as a common mutation across datasets/models for comparison purposes. Some other mutations showed similar behaviour, yet some led to most of the posteriors being narrow and centred close to 0 or 1. It is, for instance, the case of *model-level* mutation of *UE*, probably because we picked by default the first layer to apply the mutation, as the exploration of parameters of a given mutation is out of the scope of the study. In that case, there is less to discuss as mutations are almost always detected/not detected. Nonetheless, it still has some relevance as it confirms the results we would get from applying MT, with the added benefit of getting some certainty over the decision. Moreover, the fact that the phenomenon occurs on some mutation operators and not on others, without the possibility of knowing it in advance, motivates our approach capacity of quantifying the *likeliness* of detecting or not a mutation.

Regarding RQb_2, the mutation magnitudes were just chosen based on their "effect" to have "low", "medium", and "high" magnitudes (for instance, 3.1, 9.29, and 30.93 for $TRD-MN$) to explore from. Yet, the same conclusions we summarised in Findings 4 and 5 of Section 5.4.3 can be applied to all mutations we used.

5.5.2 Effect analysis

In Section 5.4.3, we have shown that using a deterministic test over a set of instances will not offer stable results for MT, hence the flakiness issue we mentioned. On the contrary, using PMT to calculate the posterior distribution of the test allows for better insights into the mutation operators under test. One crucial point is that, as mutations can be both detected and not detected by a test for some particular instances, the notion of killing a mutation, as used in traditional MT, does not seem to be very relevant in the context of ML. As such, we prefer to refer to the notion of mutation being *likely* detected, i.e., do we have sufficient evidence in a direction to assert it, similarly to how *effect size* criterion would be used in a

statistical test.

One consequence of this decision led us to introduce the similarity ratio metric $\mathcal{R}$, which allows for practical decision over the calculated posterior. We showed in Section 5.4.3 that this metric yields a more insightful and finer-grain analysis than the simple binary outcome of current MT frameworks. In particular, with PMT, we can get stable and coherent test results over the mutation operators, which would not be the case with MT because of the flakiness stemming from the selection/choice of the instances. We proposed an empirical scale to quantify the effect given by $\mathcal{R}$, based on scales used for *effect size* in statistical testing. This scale gives the user a rough idea of the confidence level attached to *how likely* the mutation is detected, which can help the user decide whether or not to consider the mutation *likely* detected. In practice, a conservative choice would be to accept only mutations with a $\mathcal{R}$ with a *strong* (or at most *medium*) effect, which results in posterior distributions being quite similar to the ideal mutant posterior. For instance, taking the mutation $MN - TRD$ we used in the motivating example of Section 5.2, using the results from Table 5.4, we would consider only the mutation of magnitude 30.93 to be detected, as the effect is *very strong*. 3.1 would be considered *likely* not detected by our test set, and the rest of the mutations being, by default, considered as not detected in order not to have false positive, even though there might be more or less evidence in a direction (small evidence that 18.57 is detected but not much information about 9.29 and 12.38).

Note that, aside from a ratio $\mathcal{R}$ around 1 or below (above) 0.82 (1.22), intermediate levels of the scale might be regarded as arbitrary. Yet, they serve the practical purpose of allowing us to be at least able to compare the effect of the different mutations in a more meaningful way than the binary outcome of MT and in a more direct way than the more complicated analysis of the posterior distributions obtained through the bagging process.

5.5.3 Trade-off PMT vs MT

Compared to MT, PMT allows obtaining a stable posterior, is fully automated and can be easily adapted to any new mutation/models using our provided framework inside the detailed replication package [200]. Our results have been computed with 200 instances, and we found the error to be relatively small for a sample size of > 190 when empirically verifying the MCE over the estimates. In practice, fewer instances may suffice depending on the precision required and the mutation operator/model under test. Note that while we compared MT and PMT using the exact same approach as proposed in DeepCrime, as it is the latest iteration of MT, one could use any approach as long as it fits the scope introduced in Section 5.3.

Nonetheless, computationally speaking, MT is less expensive than PMT, which requires

more instances to calculate the posteriors. For example, in our case, if training 20 instances to apply MT takes T seconds, our method will require $10T$ seconds (not accounting for potential parallelization). We neglect the bagging calculation time as it is small compared to the actual training time since it is independent of the model considered and depends only on the number of bootstraps and instances. For the same number of instances, a more complex model will increase the training time yet will not affect the bagging calculation. As such, in time-constrained settings, simple MT can be more beneficial in getting an answer. Note that it is possible to lower the number of instances required for PMT, yet it comes at the cost of the stability of the posterior obtained, which will depend more on the sample choice, as shown in RQb_2. Simply put, the fewer instances are used in PMT, the closer we get to MT application (and so less time is needed to get an answer) yet the more prone to flaky effects we are. In any case, the choice of the balance between time and error is left to the user and is application-dependent, based on the level of precision required. We believe the ability to better analyze mutations in DNN settings (in particular, to avoid potential tests yielding that a mutation is killed when it's not) outweighs the increase in cost due to the higher number of instances needed. This is especially true for DNNs used in safety-critical systems, where the reliability of tests is crucial. As such, we think PMT is a positive direction for improving MT application.

5.6 Threats to Validity

Construct validity. PMT relies on some approximations in which error is empirically evaluated. As such, there is an intrinsic error that we cannot reduce to theory and which depends on the model/dataset/mutation used. The rest of the assumptions are grounded in theory or previous research works. However, we showed empirically that for a sufficient number of instances, the error is relatively small and thus does not impact the decision much.

Regarding the empirical scale, if it is mainly based on our experimental results, its primary purpose is to allow us to draw a fair comparison among different mutations to assess their different effects. Moreover, the scale was designed to be a direct way of interpreting and comparing the posterior distributions. Our approach was built to be more practical for the user, and so the absolute value of the scale is less important than the relative comparison we can draw from it.

Internal validity. Because of the computation overhead induced by our method, many instances being needed for each mutation operator, we had to choose which mutation to evaluate and which model. As such, the choice of the mutations and models could impact the results. To mitigate this threat, we chose mutation operators based on DeepCrime's Killability /

Triviality analysis performed for their mutation operator. We used mutation operators listed in both DeepMutation and MuNN for Model-level mutation. Regarding mutation parameters, we used parameters provided in the replication package of DeepCrime [116] and the one mentioned in MuNN [114]. We also kept similar mutations across models/datasets to allow for a point of comparison.

For the specific choice of models/datasets, we chose the models/datasets used in DeepCrime as they were provided in their replication package, along with mutation operators designed to work on such models, which is more practical and ensures better replication ability. The particular choice of models/dataset was then motivated by the number of epochs to reduce the computation overhead. Still, we made sure to choose diverse enough subjects (regression and classification, image-based and non-image-based, etc.) to improve generalization.

Finally, note that although we have used the same mutation test as DeepCrime because it is the latest MT approach designed and because we leveraged their instances for comparison, the mutation function could be anything the user deems fit, as long as it respects the definition provided in Section 5.3. Thus, while more research should explore different MT approaches, PMT is general enough to apply in various scenarios.

External validity. We chose the same models/datasets as in DeepCrime, based on the popular framework Keras, which may limit the generalization of the study. Yet, it was necessary to ensure that we used mutations, models, and datasets in the same way as DeepCrime to draw a fairer comparison as we used their instances, mutation detection definition, and some of their operators. Nonetheless, we expect the results to generalize because the process is independent of the models/datasets/mutations used.

Reliability validity. Not to overwhelm the paper with results, and due to space limitations, we did not include all the results of our experiments. Therefore, we have provided the complete results in our replication package. We also provide all the details required to replicate our study and the implementation of PMT in our replication package [200].

5.7 Chapter Summary

This chapter introduced PMT, a probability framework for MT in DL using Bayesian probability and Bayesian Bagging, to solve the flakiness inherent to current MT approaches, which we hinted at in the previous chapter and showed in this chapter. Using real-faults-based mutations, we evaluated PMT, showing how to leverage it to decide whether a mutation can be detected more reliably than previously proposed. In particular, we highlight, for instance, that 25% of the mutations detected by the previous MT definition should not be so in light

of the probabilistic analysis offered by our framework. Moreover, we showed that for 100 instances, the error in the approximations made in PMT (approximate bagged posterior and sample size effect) is relatively small, effectively stabilizing the posterior obtained by the process. Finally, the approach is fully automated, with the decision not inducing a huge time overhead once instances are trained. Furthermore, the framework can be extended easily to any model/dataset/mutation. By doing so, PMT improves the trustworthiness of the application of MT in DNNs by accounting for the stochasticity of the DL process.

CHAPTER 6 GIST: GENERATED INPUT SETS TRANSFERABILITY IN DEEP LEARNING

To foster the verifiability and testability of Deep Neural Networks (DNN), an increasing number of methods for test case generation techniques are being developed. When confronted with testing DNN models, the user can apply any existing test generation technique. However, it needs to do so for each technique and each DNN model under test, which can be expensive. The objective of this chapter is to focus on the cost of test generation techniques when it comes to instances of DNN models by proposing the notion of test transfer. We show that there exists a similarity for which our approach can select the test set(s) to transfer better than random in all cases. By doing so, we can recover up to 80% of the property (e.g. faults detected) we would have had we generated the test from scratch instead of transferring. Finally, we show transfer is more effective as a ratio of property coverage/time spent when considering transfer for two DNNs in some cases and for four DNNs in all cases studied.

6.1 Introduction

In previous chapters, we focused on testing methods, specifically MT, that aimed to find faults at the specification-level of DNNs. We saw how stochasticity could affect the outcome of MT, and we described how this effect could be mitigated. In the next chapters, we instead focus on testing techniques that aim to detect faults that are at the model-level, more specifically, test generation techniques. While multiple test generation techniques have been proposed to test DNNs as we detailed in Section 3, this multitude of techniques raises multiple issues: First, by design, none of these techniques can be the best in all situations, as they assess different testing aspects using very different criteria. Thus, using multiple techniques together is preferable. Secondly, those generation techniques all require a certain time budget. This budget is dependent on the number of test cases generated and the size of the DNN model under test, and we saw in Chapter 3 that this cost is a limiting factor for such techniques. Finally, this cost needs to be paid every time a generation technique is applied for a given DNN model under test. This can quickly become time-consuming if multiple DNN models have to be tested. Therefore, we believe that this testing process would benefit from a paradigm change: instead of regenerating the test set independently for each DNN model under test, we could instead leverage transfer from existing DNN models. This would lower the cost of testing techniques. Nonetheless, while lowering the cost, we need to ensure that the transferred test sets follow similar properties, for instance, the faults detected, as

the test sets we would have obtained if we had applied any chosen testing techniques directly.

In this study, we present Generated Inputs Sets Transferability (GIST), a framework for transferring test sets based on a property the user is interested in for its testing. As the property to transfer can not be measured on a new DNN model under test, GIST works by leveraging a proxy to transfer the property of interest. Here, we refer to a proxy as any metric the user deems useful and that should correlate with the property to transfer while not relying on unavailable data during transfer. GIST works in two phases, an offline phase for pre-computation and an online phase when GIST is applied to any number of DNN models under test. During an offline phase, GIST uses available DNN models to validate the proxy. To do so, GIST uses a "leave one out" procedure on each available DNN model, simulating having a DNN model under test, to verify the correlation between the proxy and the property. This is done only once per test case generation technique. In the online phase, GIST uses the proxy to transfer the relevant test sets, using only the information of the DNN models under test. GIST being agnostic to the couple property/proxy chosen, it is left to the user to select them, though we recommend some initial proxies based on the results of our study.

The findings of this study are based on two datasets of different modalities (image and text), five different model architectures for each dataset and two different target properties. We highlight that GIST can transfer test sets between DNNs effectively, that is, when we aim to transfer on more than two DNNs test sets transferred cover the property targeted proportionally more compared to the ratio of time between applying the testing technique from scratch and applying GIST. We also study the effect of the proxy chosen, by using similarity (both representational and functional) metrics that are used in the literature to compare DNNs. To show that GIST can help improve trustworthiness in testing through the transfer, we show the process is resilient against stochasticity (e.g., the initial seed of the DNN to transfer) as well as architecture and modality (e.g., images, text etc.) through different experiments.

To structure the study in this chapter, we define four RQs:

RQc_1 *How do similarities and properties relate to each other in terms of DNN model types?* In this RQ, we aim to investigate the relationship between DL models' similarities and the properties such as faults detected by the test sets. This investigation is supported by the hypothesis that similar models should have test sets covering similar properties. Thus, we should observe higher property coverage on similar models which could vary depending on the actual similarity metric used.

RQc_2 *Can the introduced similarities be used as a proxy for the defined properties?* This RQ

aims to validate that the similarities investigated previously can act as a proxy for the properties of the test sets we aim to transfer. Indeed, as the goal is to mitigate testing techniques application, one would not have access in practice to the test set properties during the transfer. Instead, we would rely on the validated proxy to operationalize the transfer. If similarities can be used in such a way, there should be a positive monotonic relation between (some of) them and the property covered by the test sets.

RQc_3 *Can similarities be used to operationalize test transfer?* This RQ aims to show that, leveraging the defined similarities, we can not only transfer the test sets but also do so in a way that the chosen transferred test sets cover as much of the desired properties as possible compared to any other available test sets. If the transfer can be operationalized, the chosen transferred test sets based on the validated similarities should result in a higher property coverage compared to any other test sets.

RQc_4 *Is transfer with GIST more effective than generating test cases from scratch from a trade-off property covered/execution time point of view?* In this RQ, we analyze the trade-off between the covered property of the transferred test sets compared to the actual computation time it takes to apply the procedure. This measurement will be based on effectiveness ratio metrics between two quantities: the covered property of the transferred test sets and the time ratio between applying the transfer procedure and using the testing technique. This allows us to analyze the relevance of applying transferability depending on diverse scenarios. For the transfer to be useful, we should observe an effectiveness ratio above 1, illustrating applying the transfer is more effective than reusing the testing techniques.

6.2 Motivating Example

To highlight the usefulness of transferring instead of generating from scratch, consider the situation where we want to test one or several DNN models (red in Figure 6.1):

Scenario A: The user selects a generation technique to generate relevant test cases for this DNN model (red images in Figure 6.1). After applying directly the test case generation technique, the user would obtain test cases with several properties of interest for assessing the DNN model under test. For instance, it could cover certain faults of the DNN model under test. This would however require a certain cost. Notably, every new DNN model under test needs to have the test generation technique reapplied, and so for every test generation technique applied, which is time-consuming.

Scenario B: The user has access to other DNN models (blue, green, orange, violet) along with their generated test sets. Instead, he could transfer test sets from those DNN models directly. The transferred test sets should match the red test set in terms of the property of the test set the user is interested in, for example in that case the faults detected. This transfer approach relies on the hypothesis that DNN models and their generated test cases should not be treated independently. Indeed, it is likely that one can leverage existing similarities between DNN models in terms of features and architecture to reuse existing test sets. In Figure 6.1, by choosing the green and blue datasets, the user can effectively regenerate a subset of the fault types he would have had generating the red test set directly. This however required much less time as, instead of generating the test cases, the user would only have to select the most relevant ones. Moreover, this will scale with the number of DNN models the user needs to test as it avoids regenerating test sets every time.

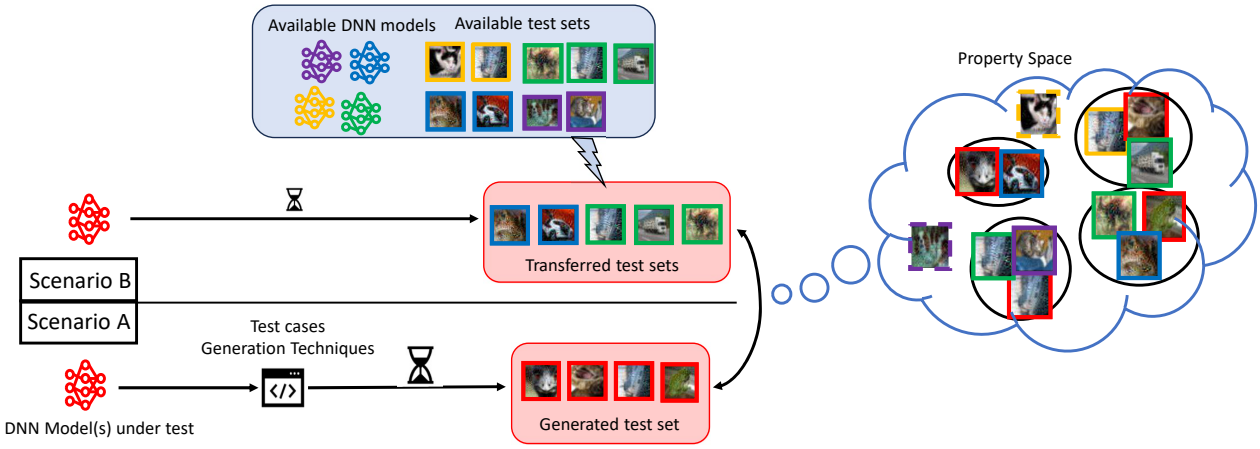


Figure 6.1 Transfering vs Generating test cases. (**Scenario A**) When generating the test cases, we apply the test cases generation technique on the DNN model under test which can be time-consuming and must be reapplied for any new DNN model under test and generation technique. (**Scenario B**) On the contrary, by transfering, we can reuse available test sets to obtain transferred test sets. Those transfered test sets should match what we would have obtained in terms of a desired property, for instance, the types of faults of the DNN model under test. Each black circle in the property space represents a cluster of fault types for the DNN model under test. Data points circled in dashed lines outside of those clusters are not faults for the DNN model under test.

6.3 Methodology

6.3.1 Test set transferability problem

Formally, we suppose we have access to a set of n reference DNN models $\mathcal{R} = \{R_1, ..., R_n\}$, all trained on the same task $\mathcal{D}$, each with generated test sets $\mathcal{T}_R = \{T_{R_1}, T_{R_2}, ..., T_{R_n}\}$. In our case, we consider that all test sets were obtained using the same generation technique $\mathbf{G}$. We consider O a DNN model under test, also trained on $\mathcal{D}$, with a test set T_O generated using $\mathbf{G}$. Note that this T_O is hypothetical in practice, as it is what we aim to match using only selected test sets from other DNN models. We want to obtain the desirable property (exactly or as close as possible) $\mathcal{P}_O$ that we would have obtained with T_O but only using available one or several T_{R_i} (that we note as T_R in the rest of the paper) from $\mathcal{T}_R$. In other words, the goal of the transfer is to find the test set(s) T_R that maximizes $\mathcal{P}_O(T_R, T_O)$. That is, the test set(s) that maximize the desirable property $\mathcal{P}_O$ to transfer relative to a hypothetical test set T_O which would have been obtained by applying $\mathbf{G}$ on O.

This, however, raises an issue: we aim to maximize $\mathcal{P}_O$ which depends on T_O which we do not have access to in practice. What we need is to find a proxy $\mathcal{P}$ which correlates with $\mathcal{P}_O$ without relying on T_O. Thus, the study and framework aim to show that it is possible to 1) recover the desirable property with the transferred test sets, 2) there exists a proxy $\mathcal{P}$ respecting the above requirements, 3) leverage the proxy to effectively recover T_O from the perspective of a given desirable property.

6.3.2 GIST: Generated Inputs Sets Transferability in Deep Learning

The proposed test set transfer approach *GIST* is composed of two phases: an offline phase during which the proxy relationship is established using available reference DNN models, and the online phase during which the test set transfer occurs on new DNN models under test. An illustration of *GIST* is presented in Figure 6.2.

During the *offline* phase (Figure 6.2 - Left), a property to transfer $\mathcal{P}_O$ is selected by the user based on requirements, and a potential proxy $\mathcal{P}$ is determined. *GIST* is agnostic to any of those. Then, the proxy relation is verified. To do so, using the sets of reference DNN models $\mathcal{R}$ and reference test sets $\mathcal{T}_R$, *GIST* samples, in turn, one of them (in Figure 6.2, on the left *(1)*) to be the pseudo DNN model under test (the *Objective DNN* model in Figure 6.2) and its associated test set (*Objective test set*). *GIST* then computes both the property $\mathcal{P}_O$ *(2)* and the proxy $\mathcal{P}$ *(3)* using the remaining DNN models as references. Note that, in that case, we can compute $\mathcal{P}_O$ since we "simulate" the fact that we have T_O by substituting a reference DNN model and its test set T_R. Finally, *GIST* evaluates the correlation for this

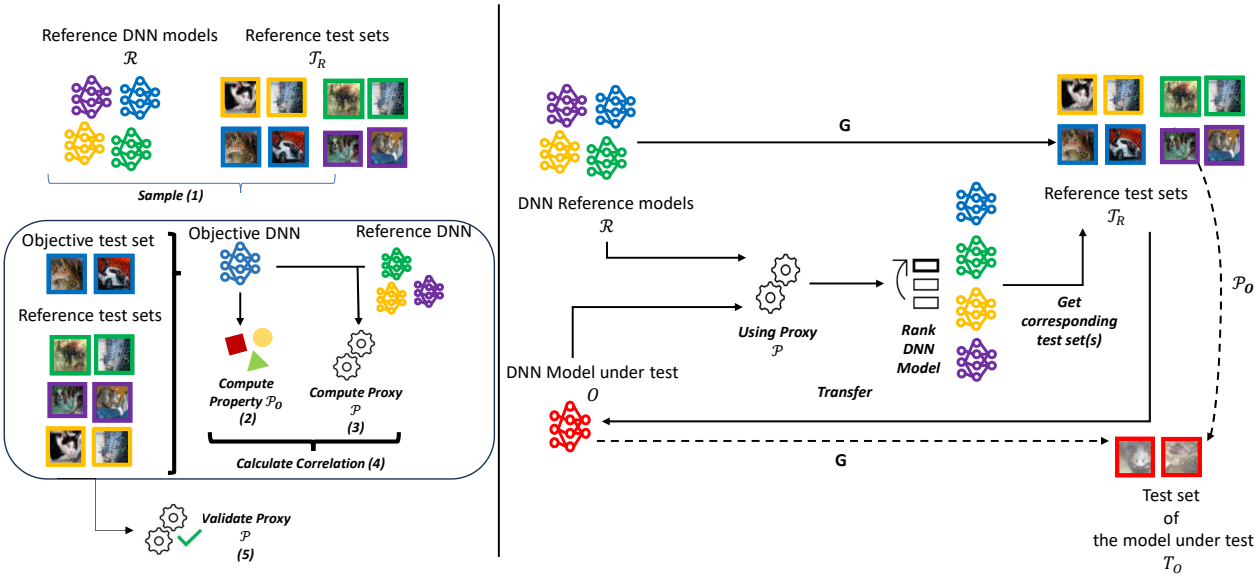


Figure 6.2 General idea of *GIST* for test sets transferability. (*Left*) **Offline** computation of *GIST* to determine the correct proxy $\mathcal{P}$. (*Right*) **Online** mode to apply on a new DNN model under test using the proxy.

objective DNN model between the property $\mathcal{P}_O$ and the proxy $\mathcal{P}$ *(4)*. This is done for all reference DNN models available in $\mathcal{R}$. In the end, if the relation holds (i.e., the correlation is deemed strong enough) for a sufficient number of DNN models, $\mathcal{P}$ is deemed to be usable *(5)*. As determining what constitutes a good proxy in this way can be complex, multiple $\mathcal{P}$ can be used simultaneously and we can retain only the best one. This process is done only once for a given generation technique **G**. We give the pseudo-algorithm of this offline phase in Algorithm 3.

Then, *GIST* can be used *online* (Figure 6.2 - Right) as many times as needed. In that setting, any new DNN model under test O is presented, a DNN model from which we do not have a T_O test set, and we wish to transfer test sets T_R from the set $\mathcal{T}_R$ on using the property $\mathcal{P}_O$. Using the proxy $\mathcal{P}$, we can rank the reference DNN models from $\mathcal{R}$ based on their distance to O. In the end, the best DNN models, according to $\mathcal{P}$, are selected and their associated test sets are transferred on O. What constitutes the "best" can be something as simple as the closest one to the DNN model under test. Yet, we will see in Section 6.5.4 that we can leverage the proxy for particular selection heuristics. Since we observed a correlation in the offline phase, we have empirical evidence that the chosen test sets will transfer effectively for the considered property. Note that, in that setting, we indeed only need the trained DNN model under test during the online phase, as everything has been determined previously in

Algorithm 3: GIST - *Offline* phase

Data: Set of reference DNN models $\mathcal{R}$, Set of reference test sets $\mathcal{T}_R$ generated using **G**, a property to transfer $\mathcal{P}_O$ and proxy to assess $\mathcal{P}$
Result: Validity of the proxy $\mathcal{P}$

```
corr ← [];
for R_i ∈ R do
    obj_DNN model ← R_i;
    ref_DNN models ← R \ {R_i};
    property ← [];
    proxy ← [];
    for R_p ∈ ref_DNN models do
        property.append(P_O(T_{R_p}, T_{R_obj_DNNmodel}));
        proxy.append(P(R_p, R_obj_DNNmodel));
    corr.append(Correlation(property, proxy));
return CheckCorrelation(corr);
```

the offline phase. We give the pseudo-algorithm of this online phase in Algorithm 4.

Algorithm 4: GIST - *Online* phase

Data: Set of reference DNN models $\mathcal{R}$, DNN Model Under Test O, set of reference test sets $\mathcal{T}_R$ generated using **G**, a property to transfer $\mathcal{P}_O$ and a validated proxy $\mathcal{P}$
Result: Reference test set(s) to be transferred $\mathcal{T}_{R_{best\_NNmodels}}$

```
proxy ← [];
for R_i ∈ R do
    proxy.append(P(R_i, O));
best_DNN models ← ChooseBest(proxy, R);
return T_{R_best_NNmodels};
```

6.3.3 Property $\mathcal{P}_O$

In our setting, the property $\mathcal{P}_O$ can be any property of the test set T_O the user aims to recover by selecting the test sets from $\mathcal{T}_R$. For example, it can be the faults detected. However, GIST is agnostic to the property required, and so any property can be used. In our experiments, we defined two properties based on two different criteria: neuron coverage and fault-types coverage.

Neuron Coverage based

To test DNN, one idea that has been introduced is to quantify how many neurons are activated (i.e., activation value is above a threshold) by a set of inputs. This echoes what is done in software testing with basic criteria such as code coverage. Pei et al. [11] introduced neuron Coverage to quantify this aspect. Later, Ma et al. [101] improved neuron Coverage by introducing k-Multi Section neuron Coverage (KMNC) as an extension of classical neuron

Coverage which aims to assess how many sections of an activation range a test set is covering in a DNN model. Formally, for a neuron n, the range $[low_n, high_n]$ of its activation for the train set of the DNN model is divided into k sections S_i^n. The i-th section is considered covered if for a given input x in a test set T we have $\phi(x, n) \in S_i^n$ where $\phi(x, n)$ is the activation of neuron n with input x. We can then define the set of all sections covered by T_i for n as $S_{T_i}^n$.

In our case, we define a property based on this metric as follows: for a given DNN model under test O, we measure the activation bands $S_{T_O}^n$ as described above. We then do similarly for each reference test set T_R, by measuring the activation bands of the inputs of T_R that lead to faults on the DNN model under test O. We obtain $S_{T_R}^n$. Finally, we can define a property $\mathcal{P}_O$ using this criteria and following our definition as:

$$\mathcal{P}_O = \frac{\sum_{n \in N} \#(S_{T_O}^n \cap S_{T_R}^n)}{\sum_{n \in N} \# S_{T_O}^n} \tag{6.1}$$

where N is a pool of neurons to be monitored and # represents the cardinality symbol. In our case, we will monitor the neurons located post-extraction layer. In other words, this property translates to the overlapping fault-inducing activation regions between our hypothetical test set T_O and any test set(s) T_R. Similarly to KMNC as a coverage criteria, the idea of the property is to quantify how many sections of the major activation regions of T_O we recover with our chosen test sets.

Fault-Types Coverage based

We saw that in the supervised learning setting, a failure in a DNN happens when, for a given test input of the test set, the DNN model's output is different from the corresponding label of the test input (see Section 2.3.2). For instance, the DNN predicts an image to be a cat when it's a dog. Failure and model-level faults are generally treated as the same thing because of the abstract nature of model-level faults in DNNs, and so one misprediction is considered as characterizing one fault. While an interesting and simplifying assumption, we might have many images of dogs in our test set that are predicted as cats. In that situation, while our test set exhibits multiple failures, and so faults with the previous associations, the faults are of the same nature which would hamper the relevance of the test set. If one were to compare it with traditional software testing, it is not necessarily helpful to cover the same branch in a program using multiple test inputs while a single test input would suffice to reveal the fault. In that case, it would be more appropriate to consider all those failures as having a unique common *type* of faults.

Aghababaeyan et al. [170] introduced an approach to cluster DNN mispredicted test inputs. They notably showed that the obtained clusters generally correlate with individual types of faults and that different clusters correspond to different type of faults in the DNN model. As such, a test set would be more desirable if it, instead of revealing a higher number of failures/faults, would reveal a higher number of *faults types*[1]. As such, to make the difference, we will refer in this chapter to *fault types* to designate the common cause leading to failures that are similar.

Their approach consists of 4 steps: 1) extracting the features over the data using an extractor and normalizing them, 2) adding two extra features (namely the predicted and original label of the test data), 3) reducing the dimensionality using Uniform Manifold Approximation and Projection (UMAP) [201] and 4) clustering the obtained embedding using Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [202]. While they use this generation technique to quantify the fault types in a test set indiscriminately, we instead propose to use it for transferability. Thus, in our case, the property we wish to transfer is simply the number of fault types of T_O that we would find in T_R when using O. Note that, by definition, only the inputs from T_R that lead to a fault in O are used in the process. With that definition, $\mathcal{P}_O$ can be defined as the proportion of overlapping faults between T_R and T_O with regard to T_O when using both test sets on O. Formally, we define a metric for faults type covered such as:

$$\mathcal{P}_O = \frac{\#(\mathcal{F}_{T_R} \cap \mathcal{F}_{T_O})}{\#\mathcal{F}_{T_O}} \tag{6.2}$$

Where $\mathcal{F}_T$ is the set of fault types of test set T and # represents the cardinality symbol.

To fit our approach, we made some changes to their method. First, instead of using a general extractor (in their case VGG19 trained on ImageNet since they only work on images), we will use the DNN model under test itself. This choice is motivated by the fact that we want to directly assess the resulting fault types on the DNN model under the test itself, not some third-party DNN model. Moreover, GIST should generalize to modalities other than images, where a general extractor such as VGG19 does not exist. Using the DNN model under test itself removes this constraint. Secondly, we make use of all the generated test sets T_R as well as T_O when creating the clusters, instead of the original test and train, as we wish to cluster the fault types on those test sets. The rest of the procedure is as we described above.

We make use of the Density-Based Clustering Validation (DBCV) [203] and Silhouette [204]

[1]Note, just like neuron coverage, we are not interested in the particular faults covered (i.e. why is fault 1 covered and not fault 2) but the number that is covered. Nonetheless, in order to keep the same expression as neuron coverage, we will talk about *fault-types* and not the *number of fault-types.*

scores to manually fine-tune the parameters of the HDBSCAN and UMAP algorithms. To validate our clustering analysis, we will verify that the obtained clusters correspond to individual fault types as shown in Section 6.5.1.

6.3.4 Proxy $\mathcal{P}$

The last step is to find the proxy $\mathcal{P}$ that we can leverage for $\mathcal{P}_O$, while only using the information that is available to us. In our problem, that information is the training data of the task D, the DNN model under test O and the reference DNN models $\mathcal{R}$ with their test sets $\mathcal{T}_R$. As we mentioned before, we do not have access to T_O in practice.

Intuitively, the more two DNN models share similar knowledge, the more they should generalize similarly. Moreover, to some extent, they should generate similar inputs especially if they are trained on the same task D and the same generation technique $\mathbf{G}$ is used on those DNN models. To get a sense of the intuition, we conducted a preliminary investigation by calculating for a diverse set of DNN models under test, in the offline setting of GIST, the properties $\mathcal{P}_O$ previously mentioned for both criteria. An example of the result is presented in Figure 6.3.

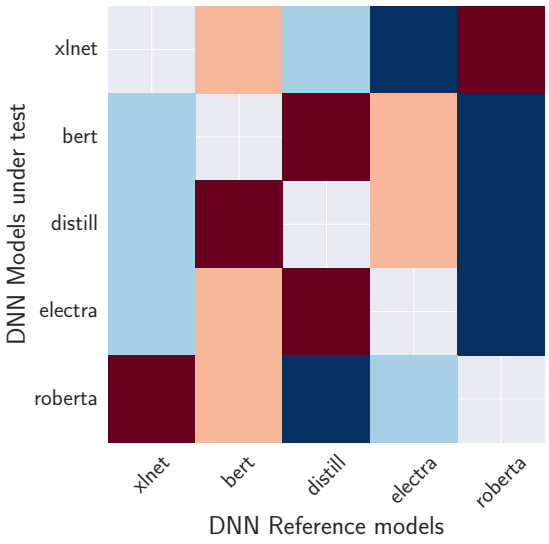


Figure 6.3 Relative $\mathcal{P}_O$ for the T_O of each DNN model under test when applying different T_R from reference DNN models. The redder and darker the colour, the higher the relative $\mathcal{P}_O$.

In that example, we used Natural Language Processing (NLP)-based DNN models with the test sets $\mathcal{T}$ being generated by some NLP generation method. More details can be found later in Section 6.4. For each DNN model under test (and so each T_O), there is generally one reference DNN model (and so T_R) which maximizes the relative $\mathcal{P}_O$ property, symbolized by

the darker red colour. The important observation is that those pairs appear to show some pattern: *BERT* will generally be paired with *Distill*, the architecture of one being derived from the other, or *RoBERTa* is generally paired with *XLNet*. As such, there seem to be some intrinsic patterns based on factors such as architecture or knowledge that seem to have an impact on how much of T_O is recovered through $\mathcal{P}_O$. Previous studies showed that there is a link between such factors and the notion of representational/functional similarities [205–207].

Thus, we suppose that said similarities could be leveraged as a proxy in our case and so we make the following hypothesis:

The more similar two DNN models are in terms of their "knowledge", the more the properties of their generated test set should overlap.

The goal of RQc_1 will be to validate this hypothesis. For the sake of describing the methodology, we assume the hypothesis to be true. In our case, as the similarities we describe in the following will be used as the proxy $\mathcal{P}$, we will refer to *similarities* instead of *proxy* when designating $\mathcal{P}$ in our experiments for clarity purposes.

The only part missing is thus to quantify this similarity. In that regard, we can leverage existing representational and/or functional similarities, that is similarities designed to quantify the "distance in knowledge" between two DNN models. Those similarities are generally split into two categories: *representational* and *functional*. Representational similarities make use of activations of pairs of layers for a fixed set of inputs. Layers in the pair can be of different sizes. On the other hand, functional similarity makes use of the output of both DNN models. Klabunde et al. [207] surveyed existing representational and functional similarities.

While for functional similarities there is no choice concerning the layer taken (the output one), it has some importance for the representational similarities. Existing approaches [208, 209] mainly used representational similarities with intermediate layers of the extractor part of the DNN models. As we mainly measure similarity between DNN models, we should aim at choosing layers with a close enough functionality across compared DNN models. The only layer with this particular property within the extractor part of the network is the last layer of the extractor, just before the classifier head. Hence, we chose this layer for representational similarities. For the set of inputs, we make use of the *training* data which were used to train both DNN models we want to compare. Using above-mentioned similarities, we can effectively obtain $\mathcal{P}(O, R)$, measuring the "similarity of knowledge" between the DNN model under test and a reference DNN model from $\mathcal{R}$. In our setting, the similarities can be used as originally described without any additional modification.

There exist different similarities, each quantifying the distance between two given DNN models differently. For instance, not all representational similarities are invariant to the same transformations over the representations (e.g., translations, permutations etc.). Since we do not know which one(s) would relate better to each property $\mathcal{P}_O$, we remained conservative and investigated six of them, three representational and three functional similarities. Those similarities were chosen as they are the most widespread in the literature. We also ensure that each distance belongs to a different similarity type according to Klabunde et al. [207]. We briefly explain how each type of similarity is computed and detail each of the selected similarities in the following. We refer the reader to Klabunde et al. [207] for detailed explanations of the similarities.

In the case of functional similarity, logit vectors are used as they are or converted to a label if needed. Regarding representational similarity, as feature vectors might be of different dimensions, we follow existing works [208] to apply the representational similarity: given feature vectors of a layer of shape (n, d) where n is the number of training data and d is the embedding dimension, said feature vectors are normalized. Then, for two DNN models M_1 and M_2, this effectively results in comparing two matrices $m_1 = (n, d_1)$ and $m_2 = (n, d_2)$. Note that, this way, we can indeed compare different architectures as the common dimension between the two matrices is the training data.

Projection Weighted Canonical Correlation Analysis (PWCCA)

Projection Weighted Canonical Correlation Analysis (PWCCA) [210] is based on Canonical Correlation Analysis which aims at finding two sets of orthogonal vectors w_1 and w_2, each of dimension d_1 and d_2 respectively, such as the projection of m_1 and m_2 onto them are maximally correlated, i.e.,

$$\rho(m_1, m_2) = max_{w_1,w_2} \frac{\langle m_1 w_1, m_2 w_2 \rangle}{||m_1 w_1||.||m_2 w_2||}$$

Each of the $k = max(d_1, d_2)$ pair of vectors $(w_1^{(i)}, w_2^{(i)})$ will lead to a canonical correlation value $\rho^{(i)}$. PWCCA is calculated as:

$$PWCCA(m_1, m_2) = \sum_i \frac{\alpha^{(i)}}{\sum_j \alpha^{(j)}} \rho^{(i)}$$

where $\alpha^{(i)} = \sum_j |\langle m_1 w_1^{(i)}, m_{1-,j} \rangle|$.

Centered Kernel Alignment (CKA)

Centered Kernel Alignment (CKA) [209] uses a kernel to calculate a $n \times n$ kernel matrix. The kernel can be any kernel function commonly used. In our case, we use a linear kernel which makes CKA be defined as:

$$CKA(m_1, m_2) = 1 - \frac{||m_1^T m_2||_F^2}{||m_1^T m_1||_F^2 ||m_2^T m_2||_F^2}$$

where $||.||_F^2$ is the Froebinus norm.

Procrustes Orthogonal (Ortho)

Procrustes Orthogonal (Ortho) [208] aims to find the best orthogonal transformation to align two matrices, i.e., $min_P ||m_1 P - m_2||_F^2$, which admits a closed form:

$$Ortho(m_1, m_2) = ||m_1||_F^2 + ||m_2||_F^2 - 2||m_1^T m_2||_*$$

Where $||.||_*$ is the nuclear norm.

Performance (Acc)

A simple similarity consists of merely comparing the performance of two given DNN models on the same dataset. Given a performance function p (i.e., evaluating how well a DNN model is doing), one can define a similarity function between two DNN models M_1 and M_2 as the absolute difference between their performance. In our case, we will use the accuracy:

$$Acc(M_1, M_2) = |p(M_1) - p(M_2)|$$

Disagreement (Dis)

Using hard predictions (i.e., the labels), one can define a similarity that is more fine-grained than simply computing the performance. Disagreement (Dis) [211] is defined as the average number of conflicts between DNN models over the predicted labels. More formally, for two DNN models M_1 and M_2:

$$Dis(M_1, M_2) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\{argmax_j M_{1_{i,j}} \neq argmax_j M_{2_{i,j}}\}$$

Divergence (Div)

Finally, using soft predictions (i.e., the logits), one can compare the probability distribution between two DNN models at the instances level. The Kullback-Leibler divergence [212] is one of the most popular statistical distances for this purpose yet it's not symmetrical. Instead, we chose the J divergence J_{Div} [212] which is a symmetrical version of the Kullback-Leibler divergence. We can then take the average of said distance overall predictions with the following:

$$J_{Div}(M_1, M_2) = \frac{1}{2N} \sum_{i=1}^{N} KL(M_{1_i}||M_{2_i}) + KL(M_{2_i}||M_{1_i})$$

Where KL is the Kullback-Leibler divergence. Note that other divergence could be used and so we refer the readers to Cha et al. [213] for an overview of possible divergence.

6.4 Experimental Design

We remind our RQs and their objectives:

RQc_1: How do similarities and properties relate to each other in terms of DNN model types?

Before showing the correlation between the similarities $\mathcal{P}$ and the property $\mathcal{P}_O$, we first set to understand the relation between similarities and measured property as well as DNN model type, that is the different DNN architecture used. The goal is to understand how similarities rank DNN model types among each other and if there is any relationship between the different similarities chosen. The goal is the same for the properties and how they relate to similarities.

RQc_2: Can the introduced similarities be used as a proxy for the defined propertie

The goal of this research question is to validate the hypothesis we detailed in Section 6.3.4, i.e., the more similar two DNN models are, the more they cover the property of the test set. To do so, we use the *Offline* procedure described in Section 6.3 and compute the correlation between the similarities $\mathcal{P}$ and the property $\mathcal{P}_O$, for each property and each technique.

RQc_3: Can similarities be used to operationalize test transfer

Building on the previous results, we aim to show *GIST* can be operationalized. Thus, we study if the similarities can be leveraged to carefully select test sets for transfer. First,

we assess simply by choosing the most similar reference test set. We are also interested in the effect of the DNN model seeds (i.e., individual initialization of a DNN model) over this choice. Secondly, we further leverage those similarities by showing they can be used to combine reference test sets in a better way than randomly sampling them.

RQc_4: Is transfer with GIST more effective than generating test cases from scratch from a trade-off property covered/execution time point of view

Finally, we measure whether transferring the test sets is more effective than simply reapplying the test case generation techniques. That is, we aim to evaluate the trade-off between property covered/relative execution time. To do so, we will measure the property the transferred test sets covered compared to a time ratio. This time ratio is the ratio between the executive time of the *Offline*/*Online* parts of *GIST* and the executive time of applying the test case generation techniques on the DNN models under test.

6.4.1 Datasets and Models

We used the following datasets and DNN models in our experiments:

Image: CIFAR10 [56] is a traditional image benchmark of 32×32 images distributed across 10 classes (dogs, cats, planes, boats etc.). The training set is composed of 50,000 images and the test set of 10,000 images. For the DNN models, we used the implementation of $VGG16$, $VGG19$, $PreResnet20$, $PreResnet110$, and $DenseNet100bc$ described in this repository [214].

Text: Movie Reviews [215] is a dataset of short positive and negative reviews from the movie reviews website Rotten Tomatoes. The dataset is split into 8,530 training data and 2,132 validation/test data. For the DNN models, we used the implementation of $Electra - small$, $XLNet$, $DistilBERT$, $BERT$, and $RoBERTa$ from HuggingFace [216].

We trained the DNN models using the parameters given in our replication package [217] and retained the weights that led to the best results on the original test set. For each DNN model type, we trained 10 DNN model seeds, i.e., 10 different initializations of the same DNN model type, to account for the effect of the stochasticity over our approach. Accuracy for the Train and Test sets are shown in Table 6.1. The choice of DNN models was motivated by the need to have a diversity of DNN models' types while keeping the DNN models' size manageable to reduce training and test generation time.

Table 6.1 Train and Test accuracy on both datasets for each DNN model. Results are averaged over all the DNN model seeds with standard deviation in between parenthesis.

CIFAR10			Movie Reviews		
Model	Train Accuracy	Test Accuracy	Model	Train Accuracy	Test Accuracy
VGG16	98.54 % (0.73)	90.49 % (0.54)	Electra	87.39 % (0.34)	82.40 % (0.47)
VGG19	98.05 % (1.34)	89.96 % (1.07)	DistilBERT	89.92 % (0.59)	83.73 % (0.25)
PreResnet20	96.08 % (0.16)	89.82 % (0.19)	BERT	89.31 % (0.80)	83.94 % (0.37)
PreResnet110	98.99 % (0.09)	91.47 % (0.16)	RoBERTa	90.91% (0.73)	87.52 % (0.45)
DenseNet100bc	99.78 % (0.06)	93.13 % (0.15)	XLNet	91.15 % (1.09)	87.01 % (0.31)

6.4.2 Test Input Generation techniques

To generate the test sets of the reference DNN models/DNN model under test, we used the following generation technique **G** depending on the dataset we were considering:

Image: Using the classification of techniques presented by Riccio et al. [218], we selected two methods: one from their *Raw Input Manipulation* category and one from their *Generative Deep Learning Models* category. We could not use techniques from their *Model-based Input Manipulatio*n category as they do not apply to CIFAR10. For *Raw Input Manipulation*, we could have picked methods such as DLFuzz [85] or DeepXplore [11]. However, we preferred to rely on a newer fuzzing technique that does not rely on neuron coverage for the fuzzing process which has shown to have several shortcomings [152,219]. We made use of FolFuzz [12] which uses First Order Loss for the fuzzing process. We leverage their technique, keeping the default parameters mentioned in the paper but just reducing the ϵ value, which controls how far the fuzzed example is from the original seed to 0.05. In doing so, the generated test cases stay very close to the inputs used to generate them. For the *Generative Deep Learning Models* category, we leveraged the Feature Perturbation-based method by Dunn et al. [84] that relies on GAN. As the technique makes use of a BigGAN architecture for the test input generation process, we use the MMGeneration library [220] implementation and checkpoints of the CIFAR10 BigGAN. For both methods, we generated 1,000 test inputs for each DNN model listed in the previous subsection to have a representative set. To avoid bias, the same starting test inputs were used for the generation in both methods so the test set will mainly differ because of the generation process (and so the DNN models used) rather than the starting inputs used.

Text: For the Text dataset, we make use of the TextAttack library [221]. Multiple methods

are implemented which make use of similar properties (swapping words, adding letters etc.). We chose to use one of the latest generation methods TFAdjusted [140] which, contrary to other methods, has shown to be better at preserving grammatical and semantical similarity. We use the provided implementation with the default parameters. As TFAdjusted will not generate a lot of misclassified inputs (because of the added constraints to ensure similarity preservation), the 2,000 base inputs of the validation set won't be enough to generate a big enough test set. To mitigate this issue, we scrapped the Rotten Tomatoes website for movie reviews in the period 2014-2022 (different from the base dataset released in 2005 to avoid overlaps) to extract reviews similar to the base dataset. We then used the ratings (0-5 scale) from the users to label them as negative or positive. To be conservative on the labelling, we consider reviews to be negative if they receive a score strictly below 2.0 and to be positive if they receive a score strictly above 3.5. Reviews with ratings not following the constraints were discarded. This allowed us to gather around 9,000 inputs we could leverage for the TFAdjusted method. Accuracy on those inputs was consistent with the test accuracy obtained on the original Movie Review dataset. Applying said method on the extracted inputs returns around 400 test-generated inputs for each DNN model. Data are available in the replication package [217].

6.4.3 Features Extraction for Similarity

As we mentioned in Section 6.3, to apply the similarity for the representational metrics, we need to extract features over some layer that has a close functionality across DNN models for the comparison to be relevant. Intuitively, among the inner layers, the layers before the classification head are good candidates for such tasks. In our case, and based on the DNN models we had to compare, we used:

Image: The output of the final pooling layer. This pooling layer makes the interface between the feature extractor and the classifier head and is present across all classifiers. As such, it makes a good candidate to evaluate their similarity.

Text: The pooled *[CLS]* token. Similar to image-based DNN models, the text-based DNN models pool the embedding obtained by the feature extractor for it to be fed to a classification head. This pooling operation is generally done by taking the *[CLS]* token's last hidden state. This token, generally added at the start of the sentence, acts as a summary of the semantics of each sentence thanks to the pretraining transformers undergo. As such, to have a relevant comparison, we took the last hidden state of each

DNN model associated with the *[CLS]* token. Note that for some transformers, such as $XLNet$, said *[CLS]* token is situated at the end of the sentence and so we made the necessary adjustments when needed. Previous work made use of a similar approach to compare transformers [222].

6.5 Experimental Results

6.5.1 Clustering and validations

Before using the fault types criteria, we validate the clustering strategy still relates to fault types since we changed the original setting by Aghababaeyan et al. [170]. To obtain the fault types of each test set on each object DNN model we applied the clustering method described in Section 6.3, by considering each individual DNN model as a DNN model under test while all other DNN models were considered reference DNN models. When clustering, we only used data from one generation technique at a time. We excluded from the reference DNN models all DNN models of the same type as the DNN model under test, e.g., if the DNN model under test is a *DenseNet100bc* trained with seed 0, then we did not use any *DenseNet100bc* trained with any other seed as part of the reference DNN models. The reasoning for this is that the test sets transfer procedure is supposed to be used when a new DNN model is being trained for which we have no reference. As such, using the same DNN model only differing by the seed is pointless in our case, as it defeats the purpose of having to transfer a test set without applying the generation technique **G** on the DNN model under test.

We obtained DBCV and silhouette scores ranging from around 0.4 to 0.8 in all cases. The scores primarily differed between generation techniques and remained relatively similar for a given DNN model type across different DNN model seeds. The lowest scores were obtained when clustering data on GAN generated test sets, which can be expected as this technique relies on GAN-generated images whose distribution is not strictly similar to the trained data. Moreover, the generation technique modifies *features* of the images rather than pixel values as is the case with fuzzing or swapping words as is the case with TFAdjusted. This would thus lead to more differences between data and make clustering harder. We included the hyperparameters used and the score obtained for each DNN model in our replication package [217] for further details.

As we mentioned in Section 6.3, we verify that clusters correspond to fault types, following a similar procedure as in the original method of Aghababaeyan et al. [170]. We picked two DNN models per generation technique of test set generation and evaluated the five biggest clusters. To do that, we retrained the DNN model [169, 223] using the original training data

and 85% of the data of a cluster. Said data are generated using each of the generation techniques and do not overlap with the train data. We then evaluate the accuracy on the remaining 15% of said clusters as well as other clusters. In a nutshell, to choose the 15 and 85 percentages we strictly followed the same method as Aghababaeyan et al. [170]. We did so three times for each cluster selected to verify that the split choice did not affect the results much. We expect that different clusters should represent individual fault types. This would translate to the accuracy of the chosen cluster, from which we used part of the data to retrain the DNN model, to be higher than on other clusters. We did not expect a perfect separation of fault types because of the limitations of the clustering techniques as well as the noise introduced in the test sets by the test generation technique. Moreover, since the same starting test inputs were used to generate the different reference test sets, the same starting inputs could lead to generated test sets clustered in different fault types (because they have a different impact on the DNN model under test). However, when retraining the DNN model with those inputs, they could affect each other as they were generated using the same input even if they differ in fault types. Results are presented in Table 6.2.

Table 6.2 Accuracy when retraining for the cluster from which data was used for retraining and for all other clusters. Results are averaged over three splits.

	DNN	Cluster C_i	Accuracy on the cluster C_i	Average Accuracy on the other cluster $C_{j\neq i}$	Model	Cluster C_i	Accuracy on the cluster C_i	Average Accuracy on the other cluster $C_{j\neq i}$
Fuzz	DenseNet100bc	Cluster 1	85.19%	10.32%	VGG19	Cluster 1	70.37%	8.25%
		Cluster 2	74.07%	10.29%		Cluster 2	45.24%	9.88%
		Cluster 3	82.05%	9.69%		Cluster 3	71.11%	12.15%
		Cluster 4	100.00%	6.59%		Cluster 4	70.59%	10.49%
		Cluster 5	90.91%	9.65%		Cluster 5	71.43%	9.54%
GAN	PreResnet20	Cluster 1	33.33%	6.30%	VGG16	Cluster 1	58.97%	7.70%
		Cluster 2	38.89%	8.87%		Cluster 2	74.07%	9.17%
		Cluster 3	94.44%	9.01%		Cluster 3	57.58%	11.23%
		Cluster 4	46.67%	5.63%		Cluster 4	42.86%	7.85%
		Cluster 5	42.86%	6.33%		Cluster 5	52.38%	9.70%
Text	RoBERTa	Cluster 1	100.0%	33.33%	DistilBERT	Cluster 1	85.71%	36.52%
		Cluster 2	100.0%	39.88%		Cluster 2	95.83%	25.07%
		Cluster 3	100.0%	32.71%		Cluster 3	100.0%	54.86%
		Cluster 4	93.33%	49.63%		Cluster 4	83.33%	37.61%
		Cluster 5	83.33%	8.03%		Cluster 5	92.31%	44.89%

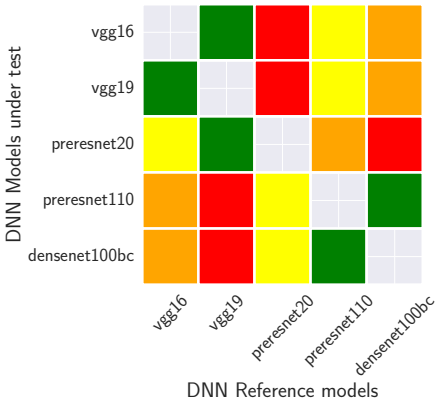
The results over the *Fuzz*-based generation technique have the most striking difference between accuracy in/out clusters, which can be explained by the fact that this technique applies low-level perturbations and so the fuzzed image will match more closely with inputs from the original test set than for instance the GAN technique which generates images from noise. The accuracy for the other clusters is generally the highest for the *Text*-based generation technique which we assume can be because the dataset only has two classes. Indeed, the noise introduced by the data from any clusters might make it easier to flip to the opposite

(and so correct) class, even if the data do not belong to the same cluster. Moreover, we did not expect perfect separation, as it is likely that inputs do not exhibit just one fault type but share multiple ones in common. Finally, contrary to Aghababaeyan et al. [170], we did not make use of the original test set in the clustering, but we made use of the test set obtained using some input generation technique, which ultimately introduced additional noise into the data. Overall, we note that the accuracy of the clusters drastically improved compared to the accuracy of other clusters. This points towards clusters representing individual forms of fault types, thus validating our fault types clustering approach.

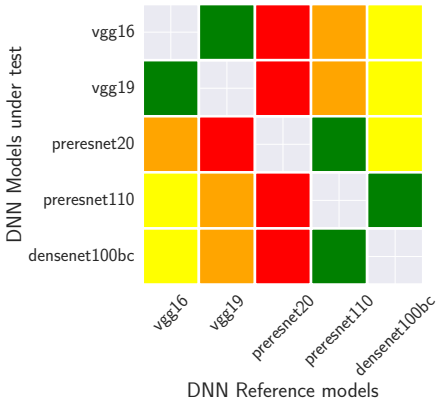
6.5.2 RQc_1: How do similarities and properties relate to each other in terms of DNN model types?

To study the relation between similarity and property, we first extract fault types and neuron activation on a given DNN model under test for each test set (all the reference test sets and the own test set of the DNN model under test). We then calculate $\mathcal{P}_O$ using Equations 6.1 and 6.2. Then, we calculate the pairwise similarity, between the DNN model under test and each reference DNN model, using each of the similarity metrics described in Section 6.3.4. Then, we averaged the obtained $\mathcal{P}_O$ for each property for a given pair (type of DNN model under test/type of reference DNN model) across the DNN model seeds. Similarly, we averaged the obtained $\mathcal{P}$ for each similarity metric for a given pair (type of DNN model under test / type of reference DNN model) across the DNN model seeds. Finally, we rank the obtained value for both metrics from 1 (most similar / covering) to 4 (least similar / covering). In this part, we are interested in their relative value to understand the underlying mechanism. Results are presented in Figure 6.4 for the *Fuzz*-based and *GAN*-based generation techniques and in Figure 6.5 for the *Text*-based generation technique.

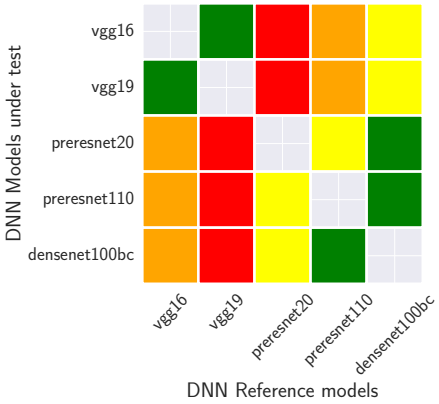
The way the heatmap for the average $\mathcal{P}_O$ can be read is, row-wise, "for a given type of DNN model under test, the averaged rank in terms of the property covered by a given reference test set is *colour*". For the similarity heatmap, it can also be read row-wise as "for a given type of DNN model under test, the given similarity rank reference DNN model as *colour* on average" where the colour indicates the rank (green = 1^{st}, red = 4^{th}). As one can see across the figures, there is some link between the rank (and so percentage) of the property covered and the DNN model type used. For instance, in Figure 6.4, both *VGG* type DNN models generally lead to more of the property being covered when we compare their T_O/T_R than the rest. Similarly, *PreResNet110* as well as *DenseNet100bc* have more covering properties compared to the rest. Analogous observations can be made for the *GAN*-based generation technique. For the *Text*-based generation technique, *RoBERTa* and *XLNet* generally have more covering properties



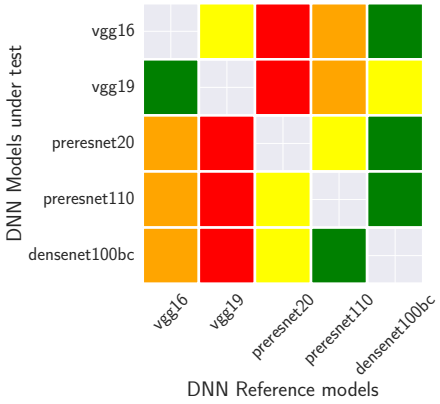
(a) Fault types - *Fuzz*-based generation



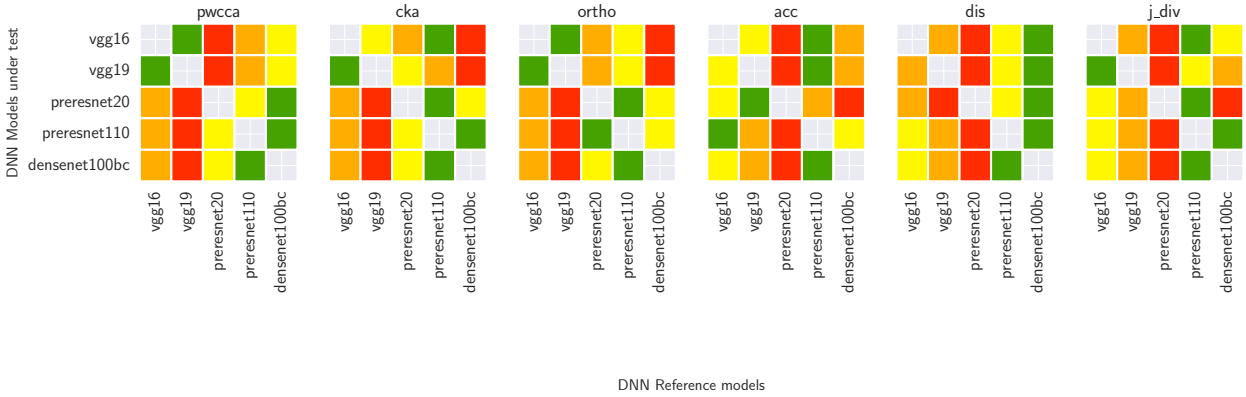
(b) Neuron coverage - *Fuzz*-based generation



(c) Fault types - *GAN*-based generation

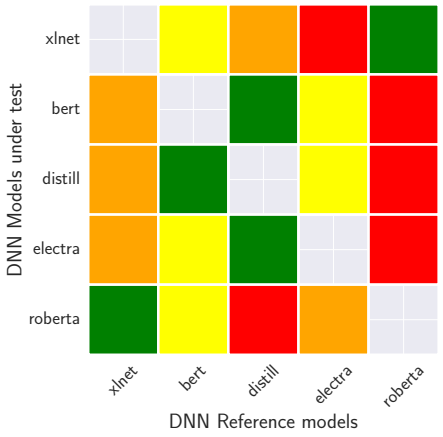


(d) Neuron coverage - *GAN*-based generation

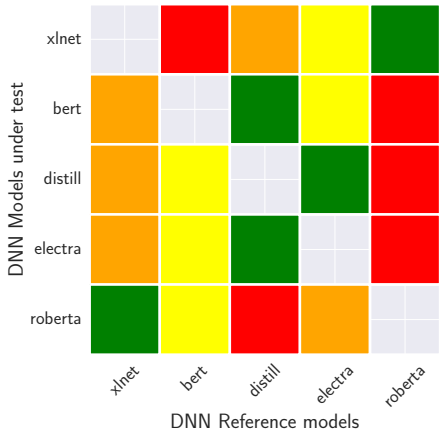


(e) Similarities

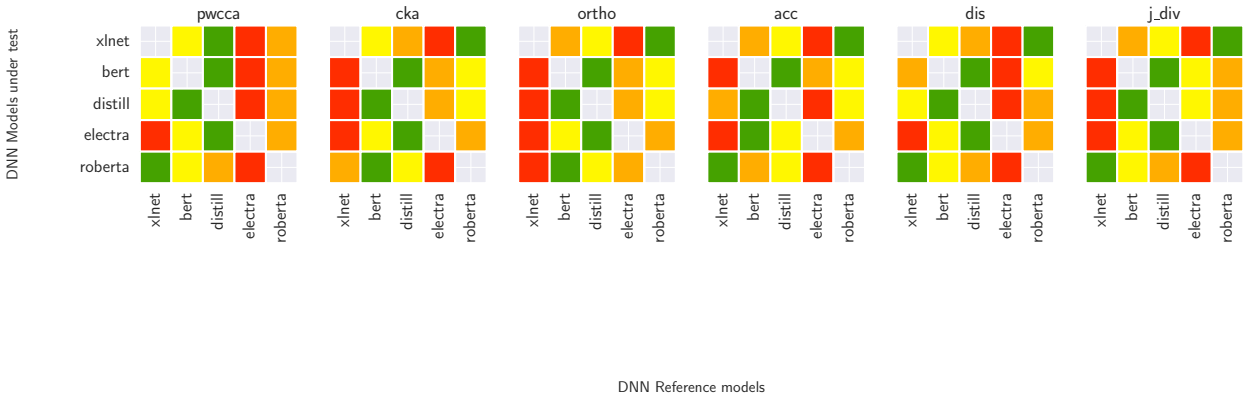
Figure 6.4 Average ranking of closest reference DNN model types in terms of $\mathcal{P}_O$ and similarity on image dataset according to a given DNN model under test. Colours indicate ranking (green $= 1^{st}$, red $= 4^{th}$).



(a) Fault types - *Text*-based generation



(b) Neuron coverage - *Text*-based generation



(c) Similarities

Figure 6.5 Average ranking of closest reference DNN model types in terms of $\mathcal{P}_O$ and similarity on text dataset according to a given DNN model under test. Colours indicate ranking (green $= 1^{st}$, red $= 4^{th}$).

while *BERT*, *DistilBERT* and *Electra* have more covering properties. Note in that case that *BERT* and *DistilBERT* have generally more covering properties than with *Electra* which is logical as *DistilBERT* is obtained by distilling a *BERT* DNN model. While heatmaps are similar between properties (fault-type and neuron coverage-based), there are still some differences: thus, for the *Fuzz*-based generation technique and $PreResnet20$ DNN model under test, it is VGG DNN models that are ranked better for the fault-type property (Figure 6.4a), while for the neuron property, it is $PreResnet110/Denset100bc$ (Figure 6.4b). Thus, it is likely that the best-fit similarity will not be the same in all situations.

If we compare the patterns obtained with the similarity heatmap, we can already infer that potential correlations can happen. As an example, for the fault-type property for the *GAN*-based generation technique (Fig 6.4c and 6.4d), we see that the ranking of $\mathcal{P}_O$ closely follows the one made by $PWCCA$ on all DNN model under test (Figure 6.4e). Thus, it is likely that, for this setting, there will be a correlation. Similarly for both properties for the *Text*-based generation technique (Fig 6.5c), J_{Div}. On the contrary, similarities such as *Acc* exhibit patterns that are less aligned with the obtained $\mathcal{P}_O$ heatmaps, which means they are less likely to lead to positive correlations.

Looking at the similarity heatmap ranking (Figure 6.4e and 6.5c), we see that the focus on what are considered two similar DNN models is not the same. On representational similarities, *Ortho* (Procrustes Orthogonal), and CKA to a lesser extent, tend to focus on the architecture of the DNN models when making the comparisons compared to $PWCCA$ or functional metrics. Thus, for image-based data, the VGG DNN model types on one side and the $PreResNet$ DNN model types on the other side will tend to be linked together while for text-based data the "BERT" family will be grouped (RoBERTa, BERT and DistillBERT) together. On the contrary, DNN model types such as *XLNet* are considered very different, having indeed a different architecture compared to "BERT" family DNN models. On the other hand, $PWCCA$ seems not to rely only on the architecture, which nonetheless seems to make it more aligned with the properties. Those observations on these representational similarities echo previous studies [205]. Functional similarities are more dissimilar among each other in terms of ranking, with *Acc* being more different than Dis and J_{Div}. Interestingly, J_{Div} seems to better match the heatmap patterns of the representational similarities, showing potential alignments on those similarities. For instance, on the *Text*-based generation technique, J_{Div} closely follows *Ortho* on all but $RoBERTa$ DNN model type. As there is no consensus across the metrics on the DNN model type, showing that similarities do rely on different aspects to quantify a similarity, studying multiple similarities for correlation seems to be a sound approach.

Findings 1: Similarities rank DNN model types differently based on their characteristics: for instance, representational and functional similarities will not necessarily consider the same DNN model types as most similar. While overlap exists, there is no agreement between similarities in any of our property/generation technique couples. This suggests that exploring multiple similarities as the proxy is important. Finally, $\mathcal{P}_O$ tends to be better covered by reference test sets associated with specific DNN model types. These patterns overlap with certain similarity ranking patterns suggesting those specific similarities could be leveraged as a proxy, in our case $PWCCA$ and J_{Div}.

6.5.3 RQ$_{c_2}$: Can the introduced similarities be used as a proxy for the defined properties

After analyzing similarities/property patterns, we aim to verify that (some of) the similarities we chose ($\mathcal{P}$) are a correct proxy for i.e., property we wish to transfer (i.e., $\mathcal{P}_O$). To do so, similarly to RQc_1, we compute both $\mathcal{P}$ and $\mathcal{P}_O$ following the offline procedure described in Figure 6.2 with each DNN model at our disposal is considered in turn as a DNN model under test. We then assess the correlation between those quantities using Kendall τ [224] as recommended to evaluate such metrics [208]. Results are presented in Table 6.3 for the $\mathcal{P}_O$ based on fault-types and in Table 6.4 based on neuron coverage. As each DNN model seed of each DNN model type is used as a DNN model under test, for each generation technique, we calculated the median, the 1st quartile as well as the 3rd quartile for Kendall τ across all DNN model seeds. We also count the number of times the correlation is significant at the 0.05 and 0.1 threshold across all DNN model seeds. Note that, in our case, we do not need to consider any correction over the p-value, such as the Bonferroni correction, as we do not perform multiple comparisons but consider the test individually [225]. Results are given in Table 6.3 for the fault-type property and in Table 6.4 for the neuron property.

Table 6.3 Correlation for each generation technique for each given DNN model being considered as a DNN model under test for $\mathcal{P}_O$ based on Fault Types. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We highlight in bold the highest results. Generation techniques used are (top to bottom): *Fuzz*-based, *GAN*-based and *Text*-based.

DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.44	0.45	**0.46**	0.29	0.31	0.38	**0.43**	0.35	0.26	0.03	0.28	0.31	-0.22	-0.07	-0.07	**0.25**	-0.22	**0.25**	**0.55**	0.17	0.37	0.43	0.29	0.50	**0.54**	0.19	0.34	0.30	0.14	0.28
tau-κ (1^{st}-Q) ↑	0.41	**0.43**	0.42	0.26	0.28	0.36	**0.35**	0.28	0.12	0.00	0.23	0.26	-0.34	-0.11	-0.11	0.20	-0.27	**0.21**	**0.51**	-0.03	0.32	0.32	0.26	0.37	**0.46**	0.09	0.31	0.24	0.10	0.16
tau-κ (3^{rd}-Q) ↑	0.47	0.48	**0.50**	0.36	0.37	0.45	**0.45**	0.42	0.27	0.16	0.34	0.36	-0.17	0.03	0.05	0.30	-0.15	**0.34**	0.57	0.20	0.40	0.49	0.32	**0.59**	**0.57**	0.27	0.35	0.35	0.21	0.41
# p-value < 0.05	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**90%**	**90%**	60%	20%	80%	80%	50%	10%	10%	60%	50%	**70%**	**100%**	30%	80%	**100%**	**100%**	80%	**100%**	40%	**100%**	80%	30%	70%
# p-value < 0.1	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**90%**	**90%**	60%	20%	80%	**90%**	50%	10%	20%	**90%**	70%	**90%**	**100%**	50%	80%	**100%**	**100%**	80%	**100%**	70%	**100%**	**100%**	30%	70%
DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.32	0.33	**0.34**	0.12	0.13	0.16	**0.48**	0.44	0.27	-0.14	0.23	0.26	**0.27**	0.14	0.17	-0.25	0.25	-0.12	**0.28**	0.07	0.16	0.09	0.15	0.22	**0.25**	0.12	0.10	0.03	0.13	0.17
tau-κ (1^{st}-Q) ↑	0.30	**0.32**	**0.32**	0.07	0.07	0.10	**0.46**	0.39	0.21	-0.18	0.19	0.19	**0.20**	0.11	0.15	-0.30	0.18	-0.18	**0.23**	-0.03	0.14	0.00	0.08	0.06	**0.19**	0.02	0.03	-0.02	0.07	0.05
tau-κ (3^{rd}-Q) ↑	**0.36**	0.34	**0.36**	0.19	0.18	0.20	0.51	**0.52**	0.31	-0.01	0.27	0.29	**0.33**	0.27	0.29	-0.20	0.29	-0.08	**0.29**	0.16	0.27	0.21	0.20	0.29	**0.28**	0.18	0.14	0.13	0.15	0.24
# p-value < 0.05	**100%**	**100%**	**100%**	10%	20%	20%	**100%**	90%	70%	20%	60%	70%	60%	30%	40%	**70%**	60%	20%	**90%**	20%	40%	30%	10%	50%	**70%**	10%	10%	0%	0%	40%
# p-value < 0.1	**100%**	**100%**	**100%**	30%	30%	30%	**100%**	**100%**	90%	30%	70%	70%	**80%**	30%	40%	**80%**	70%	30%	**90%**	20%	40%	50%	30%	60%	**70%**	30%	20%	10%	10%	50%
DNN	ROBERTA						XLNet						BERT						DistilBERT						Electra					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.33	0.03	-0.23	0.31	0.49	**0.50**	-0.03	**0.47**	0.46	0.42	0.44	0.45	0.16	0.28	0.26	0.17	0.15	**0.59**	0.09	0.41	0.44	0.00	0.28	0.63	0.54	0.56	0.43	0.41	0.56	**0.62**
tau-κ (1^{st}-Q) ↑	0.27	-0.06	-0.34	0.09	0.40	**0.41**	-0.05	**0.43**	0.36	0.37	0.41	0.41	0.12	0.22	0.21	0.02	0.11	**0.52**	0.08	0.37	0.37	-0.03	0.18	**0.59**	0.50	0.52	0.41	0.39	0.54	**0.60**
tau-κ (3^{rd}-Q) ↑	0.38	0.09	-0.17	0.34	**0.58**	**0.58**	0.05	0.54	0.51	0.48	**0.57**	0.50	0.19	0.29	0.28	0.30	0.16	**0.64**	0.13	0.44	0.47	0.08	0.38	0.66	0.58	0.61	0.48	0.43	0.59	**0.67**
# p-value < 0.05	90%	0%	50%	60%	**100%**	**100%**	0%	**100%**	80%	90%	**100%**	90%	10%	70%	70%	40%	0%	**100%**	0%	90%	**100%**	10%	70%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
# p-value < 0.1	90%	0%	60%	80%	**100%**	**100%**	10%	**100%**	80%	90%	**100%**	**100%**	40%	80%	80%	40%	10%	**100%**	10%	**100%**	**100%**	10%	70%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

Table 6.4 Correlation for each generation technique for each given DNN model being considered as a DNN model under test for $\mathcal{P}_O$ based on neuron coverage. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We highlight in bold the highest results. Generation techniques used are (top to bottom): *Fuzz*-based, *GAN*-based and *Text*-based.

DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.37	0.39	0.41	0.45	0.47	**0.54**	0.37	0.29	0.07	0.25	0.61	**0.62**	0.36	0.41	**0.42**	-0.36	0.43	-0.09	**0.57**	0.11	0.33	0.43	0.36	0.54	**0.60**	0.22	0.36	0.36	0.25	0.40
tau-κ (1^{st}-Q) ↑	0.34	0.36	0.38	0.43	0.46	**0.53**	0.35	0.25	0.02	0.19	0.57	**0.58**	0.32	0.36	**0.39**	-0.39	**0.39**	-0.15	**0.55**	0.03	0.20	0.38	0.30	0.45	**0.57**	0.10	0.31	0.33	0.19	0.33
tau-κ (3^{rd}-Q) ↑	0.38	0.40	0.42	0.46	0.48	**0.55**	0.39	0.31	0.10	0.33	0.64	**0.67**	0.41	0.45	**0.49**	-0.30	0.47	-0.05	**0.63**	0.16	0.36	0.49	0.41	0.56	**0.63**	0.27	0.41	0.41	0.29	0.49
# p-value < 0.05	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	90%	80%	0%	60%	**100%**	**100%**	90%	90%	90%	**100%**	90%	0%	**100%**	20%	70%	**100%**	**100%**	90%	**100%**	50%	90%	**100%**	60%	90%
# p-value < 0.1	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	90%	80%	0%	80%	**100%**	**100%**	90%	90%	90%	**100%**	**100%**	0%	**100%**	30%	70%	**100%**	**100%**	**100%**	**100%**	60%	**100%**	**100%**	80%	90%

DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.47	0.48	**0.51**	0.13	0.15	0.19	**0.57**	0.51	0.29	-0.12	0.27	0.29	**0.42**	0.27	0.28	-0.40	0.41	-0.26	**0.34**	0.08	0.11	0.15	0.19	0.24	**0.30**	0.10	0.06	0.06	0.17	0.21
tau-κ (1^{st}-Q) ↑	**0.46**	0.44	0.47	0.10	0.11	0.16	**0.54**	0.46	0.27	-0.14	0.25	0.28	**0.37**	0.20	0.24	-0.42	**0.37**	-0.27	**0.25**	-0.02	0.09	0.11	0.15	0.12	**0.28**	0.06	0.04	0.02	0.15	0.16
tau-κ (3^{rd}-Q) ↑	0.49	0.50	**0.52**	0.16	0.18	0.23	**0.59**	0.54	0.32	-0.07	0.31	0.32	**0.45**	0.31	0.34	-0.37	**0.45**	-0.22	**0.39**	0.16	0.21	0.18	0.24	0.31	**0.36**	0.13	0.08	0.11	0.25	0.27
# p-value < 0.05	**100%**	**100%**	**100%**	10%	10%	30%	**100%**	**100%**	90%	0%	**100%**	**100%**	**100%**	60%	**100%**	**100%**	**100%**	70%	**80%**	20%	30%	20%	40%	70%	**100%**	0%	0%	0%	40%	40%
# p-value < 0.1	**100%**	**100%**	**100%**	20%	20%	50%	**100%**	**100%**	**100%**	10%	**100%**	**100%**	**100%**	90%	**100%**	**100%**	**100%**	80%	**90%**	30%	40%	40%	50%	80%	**100%**	0%	0%	0%	40%	70%

DNN	ROBERTA						XLNet						BERT						DistilBERT						Electra					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
tau-κ (median) ↑	0.26	-0.03	-0.35	0.25	0.46	**0.47**	-0.10	0.38	0.33	0.36	0.37	**0.39**	0.13	0.13	0.10	0.11	0.14	**0.45**	0.06	0.28	0.30	-0.07	0.23	**0.55**	0.52	0.49	0.41	0.33	0.53	**0.55**
tau-κ (1^{st}-Q) ↑	0.20	-0.09	-0.38	0.07	**0.42**	0.41	-0.17	**0.32**	0.26	0.25	0.29	0.27	0.11	0.11	0.07	0.00	0.12	**0.40**	-0.01	0.26	0.27	-0.10	0.08	**0.49**	0.47	0.46	0.38	0.28	0.51	**0.53**
tau-κ (3^{rd}-Q) ↑	0.31	0.02	-0.32	0.35	0.49	**0.51**	-0.05	**0.45**	0.44	0.42	0.42	0.44	0.16	0.15	0.14	0.32	0.16	**0.50**	0.10	0.31	0.34	0.02	0.30	**0.57**	**0.57**	0.51	0.43	0.33	0.55	0.55
# p-value < 0.05	50%	0%	90%	60%	**100%**	**100%**	10%	**90%**	80%	**90%**	**90%**	**90%**	10%	10%	0%	40%	0%	**100%**	0%	**100%**	**100%**	10%	50%	**100%**	**100%**	**100%**	**100%**	90%	**100%**	**100%**
# p-value < 0.1	80%	0%	90%	60%	**100%**	**100%**	30%	**100%**	80%	90%	**100%**	90%	10%	20%	10%	40%	10%	**100%**	10%	**100%**	**100%**	10%	60%	**100%**	**100%**	**100%**	**100%**	90%	**100%**	**100%**

First, we observe that no metric is overall the best for all generation techniques and all DNN models under test. As such, there does not seem to be a "one-metric fit-all" similarity that could be used in every situation. This was nonetheless to be expected as, analogically to the *No Free Lunch* theorem, it is not likely that one metric can outperform all others in every situation. Especially when different generation techniques (and so different input types) are used and that the similarity focuses on particular invariance. Note that the DNN model seed can have its importance on the results, as illustrated by the variation for the same metric of the number of significant (strength of the) correlations. This illustrates the need to consider multiple DNN model seeds when doing this sort of study to quantify the effect said seed can have on the results. We also see that results drastically differ depending on the metric that is used and it's rare that all metrics lead to all correlations being significant.

However, it's possible to find, for each generation technique, one similarity that works reasonably well across all the selected DNN models under test. Regarding Fault Types based $\mathcal{P}_O$, J_{Div} is the best overall for the *Text*-based generation technique, with medium/strong correlations (> 0.4) which are significant almost across all DNN model seeds at the 0.05 threshold. For the *GAN*-based generation technique, it's instead $PWCCA$ which seems to be preferable. One can see in that case that the correlation strength largely varies across DNN models under tests (median around $\sim$ 0.25 - 0.40). For the *Fuzz*-based generation technique, the choice is between $PWCCA$ and J_{Div}: while the first one gives good results on 4 types of object DNN models (median correlation $\sim$ 0.4 - 0.5, almost all being significant), it performs relatively badly on *PreResnet20* (negative correlation). On the contrary, J_{Div} leads to a weaker correlation (median correlation $\sim$ 0.25 - 0.5, with a lesser number of one being significant), but is more consistent across DNN models.

Regarding the neuron-based $\mathcal{P}_O$, similar results can be observed for the *Text*-based and the *GAN*-based generation techniques, with respectively J_{Div} (medium/strong correlations > 0.35) and $PWCCA$ (medium correlations > 0.3). We note that the strength of the correlation has followed opposing trends: while for the *Text*-based generation technique the correlations have become relatively weaker from the neuron-based coverage, it has gotten stronger on *GAN*-based generation technique for the same type of DNN model under test. The main difference between the two properties is visible with the *Fuzz*-based generation technique: for neuron-coverage based $\mathcal{P}_O$ it is $PWCCA$ which works consistently well across DNN model types (median around $\sim 0.35 - 0.6$), while J_{Div} works well across most DNN model types except $PreResNet20$. Dis would also be a reasonable similarity in that case as it works well on all DNN model types except $VGG16$. The obtained results for both properties match observed patterns in RQc_1.

Findings 2: There is not one single effective similarity for all DNN models under tests and generation techniques. However, for each technique, we can find one similarity that leads to a positive significant correlation with each property across all DNN models under tests, for a majority of DNN model seeds used. Results can be impacted by the DNN model seed choice illustrating the need to consider this aspect in the process. $PWCCA$ and J_{Div} seem to be the main similarities that achieve good significant correlations across our experiments. In particular, for both properties (fault type and neuron based), $PWCCA$ works best for *GAN*-based generation technique and J_{Div} works best for *Text*-based generation technique. Both metrics work reasonably well for *Fuzz*-based generation technique. Those results confirm the preliminary results of RQc_1.

6.5.4 RQc_3: Can similarities be used to operationalize test transfer?

We now want to verify if the most relevant reference test set in terms of property covered (i.e., fault-type coverage and neuron coverage) can be decided from the similarity and how the DNN model seeds affect this outcome. To do so we will use two criteria: *Top-1* and *Top-5* which aim to compare the property coverage of the chosen test set against a randomly picked test set. Those criteria are inspired by the *Top-1* and *Top-5* accuracy metrics used in classification or information retrieval [226] and aim to show the relevance of a chosen test set in comparison to other test sets in terms of the property to transfer. As we have shown in the previous RQ we have several significant correlations. Thus, the most similar DNN model according to the chosen representational/functional similarity should lead to the most covering test set for the given property. Thus, as a criterion, we can choose the test set of the most similar reference DNN model to be transferred in practice. To evaluate the effectiveness of the criterion, we compare the property coverage of the chosen test set to the coverage of the remaining test sets, as if we were choosing at random. Basically, we count how many other test sets have covered more of the property compared to the selected test set. This is the *Top-1* metric. However, we have seen that the DNN model seed can affect the results so we also want to assess how the result would differ if the most similar DNN model was not selectable (for instance, because we did not train such an instance). This is done to assess the robustness of the method against seed selection. As such, we iteratively repeat the same procedure as the *Top-1* metric while removing the first k (with k iteratively being $\{1, 2, 3, 4\}$) best DNN models from the choice and choosing the $k + 1$ to become the best. This is the *Top-5* metric. Similarly, as in the previous research question, we compute the result for each DNN model seed as the DNN model under test and we show the median,

first, and third quartiles. Finally, we also provide the averaged value of $\mathcal{P}_O$ for each property across the DNN model seeds of the DNN model under test for the *Top-1* and *Top-5* metrics. Results are presented in Table 6.5 for the fault-type property and in Table 6.6 for the neuron property.

For all generation techniques and both properties, the metrics which led to the highest and most consistent correlations across DNN models under tests in previous RQs yield good results in terms of the *Top-1* and *Top-5*. However, other metrics can be better at choosing the best reference test set for some particular DNN model under test. For instance, *CKA* for the *Fuzz*-based generation technique also works well for DNN models such as *PreResnet20* or *VGG19* while *CKA* did not have a significant correlation across all DNN model seeds. As such, this shows that the absence of a correlation across the reference sets does not necessarily mean that the most similar ones are not among the most property covering sets. Nonetheless, while the chosen metric is not necessarily the best for a given DNN model, it will generally be good across all DNN models under test. Indeed, if we rank the different similarity metrics for both *Top-1* and *Top-5* metrics (1 the best, 6 the lowest), the chosen metrics will have the highest average rank compared to all others. For instance for *GAN*-based generation technique, $PWCCA$ has an average rank of 1.6 for *Top-1* criterion, where other metrics are above 2 (i.e. on average $PWCCA$ is ranked 1.6 out of 6 on the DNN models under test). We obtain similar results for J_{Div} on *Fuzz*-based and *Text*-based generation techniques both on *Top-1* and *Top-5* criteria. This is the case no matter the property chosen. Moreover, as the *Top-1* and *Top-5* values are lower than 50%, choosing the similarity that led to overall good correlations in the previous RQs is better than picking at random a reference test set.

Regarding the value of the coverage for the reference test sets of the *Top-1* and *Top-5* DNN model seeds, the delta of *Top-1* and *Top-5* values do not change much on average. This shows that top DNN model seeds' reference test sets are consistent for both covering properties and a given DNN model under test. In terms of absolute value, there are discrepancies based on the property, the generation technique and the DNN model type. On fault-types based property, the coverage range is $\sim$ 20%-45% for the relevant similarities (i.e, $PWCCA$ and J_{Div}). On the neuron-based property, the range is wider with $\sim$ 15%-45%. In particular, VGG and $RoBERTa/XLNet$ DNN model types have their T_O less covered in terms of this property. Nonetheless, by selecting one test set through our similarity, we can already cover in many cases more than a third of the covering property of T_O.

Table 6.5 For fault-type based property, *Top-1* and *Top-5* metric for each generation technique, for each given DNN model seed being considered as a DNN model under test. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We also show the averaged value of fault-type coverage property for both *Top-1* and *Top-5*. We highlight in bold the best scores. Generation techniques used are (top to bottom): *Fuzz*-based, *GAN*-based and *Text*-based.

DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	11%	5%	**4%**	54%	54%	26%	**6%**	16%	30%	51%	29%	15%	62%	17%	**11%**	88%	70%	53%	**9%**	12%	11%	11%	40%	11%	18%	10%	15%	7%	11%	**5%**
Top-1 (1^{st}-Q) ↓	6%	**3%**	**3%**	50%	50%	21%	**1%**	8%	22%	33%	16%	8%	48%	7%	**6%**	66%	63%	38%	**3%**	11%	8%	5%	36%	4%	14%	6%	10%	**3%**	8%	4%
Top-1 (3^{rd}-Q) ↓	18%	12%	**10%**	64%	64%	48%	**9%**	42%	63%	60%	34%	36%	77%	32%	**28%**	97%	88%	68%	**14%**	24%	20%	28%	67%	19%	21%	18%	23%	35%	26%	**16%**
Top-1 Value ↑	33.86%	35.11%	**35.33%**	27.20%	27.20%	30.17%	**36.88%**	34.11%	32.36%	31.94%	33.33%	34.36%	31.08%	33.80%	**34.38%**	28.07%	29.52%	31.29%	45.88%	45.52%	45.83%	43.39%	39.08%	**46.33%**	42.45%	40.30%	44.14%	38.91%	35.33%	**44.35%**
Top-5 (median) ↓	11%	**9%**	11%	47%	40%	24%	**13%**	26%	40%	46%	21%	**13%**	70%	**27%**	31%	56%	75%	40%	10%	11%	11%	29%	43%	**9%**	30%	**20%**	35%	30%	28%	**20%**
Top-5 (1^{st}-Q) ↓	10%	**8%**	**8%**	40%	36%	20%	**10%**	21%	37%	40%	17%	**10%**	65%	24%	**23%**	50%	68%	32%	9%	**8%**	**8%**	18%	38%	**8%**	27%	14%	23%	21%	22%	**10%**
Top-5 (3^{rd}-Q) ↓	13%	**10%**	14%	53%	43%	29%	**19%**	40%	50%	48%	27%	25%	76%	**35%**	36%	60%	81%	45%	**13%**	14%	**13%**	39%	49%	36%	37%	31%	51%	49%	31%	**24%**
Top-5 Value ↑	33.54%	**33.95%**	33.53%	29.35%	29.61%	32.80%	**35.14%**	33.66%	32.02%	31.58%	34.41%	34.40%	29.82%	**33.36%**	33.23%	31.42%	29.50%	32.68%	45.58%	**45.81%**	46.79%	40.93%	40.10%	43.75%	**43.90%**	39.62%	43.54%	37.74%	36.14%	40.17%
DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	**10%**	**10%**	11%	69%	69%	**10%**	**0%**	12%	39%	72%	23%	6%	**17%**	30%	33%	90%	31%	57%	**10%**	16%	11%	31%	62%	31%	35%	35%	**3%**	55%	26%	10%
Top-1 (1^{st}-Q) ↓	**1%**	3%	3%	66%	66%	3%	**0%**	11%	19%	61%	17%	1%	**4%**	18%	18%	86%	13%	36%	**2%**	10%	6%	23%	39%	12%	26%	16%	**1%**	31%	9%	8%
Top-1 (3^{rd}-Q) ↓	**19%**	**19%**	24%	75%	75%	62%	**5%**	28%	51%	79%	27%	18%	**19%**	46%	68%	97%	42%	64%	**20%**	26%	**20%**	79%	78%	57%	51%	64%	29%	69%	57%	**19%**
Top-1 Value ↑	**26.56%**	26.25%	26.09%	21.42%	21.42%	25.41%	**25.12%**	23.35%	21.78%	19.33%	22.33%	24.36%	**22.66%**	20.97%	20.69%	17.11%	21.85%	20.32%	35.40%	35.03%	**35.99%**	32.29%	32.15%	34.15%	32.67%	32.58%	**34.68%**	31.19%	33.50%	34.64%
Top-5 (median) ↓	**13%**	23%	17%	44%	41%	36%	**13%**	17%	40%	67%	17%	14%	33%	33%	**31%**	64%	42%	55%	**20%**	27%	25%	41%	49%	29%	**22%**	42%	**22%**	47%	39%	31%
Top-5 (1^{st}-Q) ↓	**10%**	19%	14%	37%	37%	32%	12%	11%	34%	65%	15%	**8%**	**25%**	30%	29%	60%	31%	52%	**15%**	18%	16%	40%	44%	23%	16%	23%	**12%**	44%	29%	28%
Top-5 (3^{rd}-Q) ↓	**23%**	28%	24%	46%	46%	42%	**15%**	32%	43%	78%	21%	18%	**34%**	44%	41%	71%	52%	59%	**28%**	29%	**28%**	56%	54%	41%	39%	56%	**31%**	56%	45%	36%
Top-5 Value ↑	**25.99%**	25.12%	25.59%	22.85%	22.76%	24.05%	**23.42%**	22.97%	21.33%	19.19%	23.21%	23.14%	**21.55%**	21.16%	21.33%	18.68%	20.99%	20.00%	**34.76%**	34.55%	34.66%	32.98%	32.50%	34.16%	33.74%	32.55%	**34.28%**	31.29%	32.61%	32.81%
DNN	ROBERTA						XLNet						BERT						DistilBERT						Electra					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	**9%**	34%	33%	28%	11%	**9%**	40%	**3%**	7%	15%	**3%**	14%	15%	**9%**	14%	44%	11%	10%	10%	19%	**9%**	53%	11%	12%	**4%**	17%	24%	34%	17%	7%
Top-1 (1^{st}-Q) ↓	**3%**	25%	30%	15%	4%	**3%**	21%	**3%**	**3%**	11%	**3%**	5%	3%	**1%**	8%	33%	7%	6%	4%	11%	**3%**	18%	8%	6%	**3%**	7%	17%	20%	8%	**3%**
Top-1 (3^{rd}-Q) ↓	**15%**	52%	47%	38%	21%	**15%**	62%	**5%**	12%	33%	**5%**	17%	17%	**10%**	19%	72%	19%	12%	16%	24%	**9%**	77%	12%	12%	22%	24%	33%	39%	29%	**10%**
Top-1 Value ↑	**35.97%**	27.15%	25.89%	29.49%	35.73%	**35.97%**	27.64%	**35.56%**	34.42%	31.01%	**35.56%**	33.23%	37.45%	**40.60%**	37.51%	26.47%	38.22%	**39.58%**	37.36%	36.04%	38.82%	27.26%	37.41%	**37.54%**	35.35%	34.18%	31.94%	30.45%	34.26%	**36.14%**
Top-5 (median) ↓	22%	40%	45%	23%	10%	**9%**	49%	**8%**	12%	13%	**8%**	**8%**	10%	**9%**	12%	38%	11%	**9%**	**8%**	10%	**8%**	40%	10%	11%	**11%**	14%	20%	37%	14%	**11%**
Top-5 (1^{st}-Q) ↓	18%	33%	27%	14%	8%	**7%**	40%	**6%**	10%	10%	**6%**	**6%**	9%	9%	10%	27%	**8%**	9%	7%	7%	**5%**	37%	7%	8%	**7%**	8%	14%	35%	10%	9%
Top-5 (3^{rd}-Q) ↓	25%	48%	55%	41%	13%	**10%**	57%	**11%**	14%	22%	12%	**11%**	11%	10%	13%	39%	12%	**9%**	**10%**	12%	**10%**	45%	12%	16%	17%	17%	31%	41%	19%	**16%**
Top-5 Value ↑	31.53%	26.24%	24.82%	29.58%	**35.22%**	35.06%	25.65%	**33.53%**	31.96%	31.30%	33.28%	33.04%	38.53%	**38.91%**	37.95%	31.02%	38.41%	38.45%	**37.67%**	37.25%	37.91%	28.95%	37.31%	36.83%	**35.34%**	35.06%	33.47%	29.26%	34.44%	35.21%

Table 6.6 For neuron-based property, *Top-1* and *Top-5* metric for each generation technique, for each given DNN model seed being considered as a DNN model under test. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We also show the averaged value of fault-type coverage property for both *Top-1* and *Top-5*. We highlight in bold the best scores. Generation techniques used are (top to bottom): *Fuzz*-based, *GAN*-based and *Text*-based.

DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	11%	**8**%	9%	41%	41%	22%	**1**%	31%	60%	34%	15%	9%	**10**%	26%	25%	98%	41%	91%	12%	20%	22%	**8**%	51%	11%	15%	24%	4%	25%	49%	**2**%
Top-1 (1^{st}-Q) ↓	**8**%	**8**%	**8**%	40%	40%	20%	**0**%	11%	56%	23%	10%	3%	**8**%	25%	16%	96%	21%	86%	8%	11%	11%	**2**%	30%	6%	12%	8%	**1**%	20%	40%	**1**%
Top-1 (3^{rd}-Q) ↓	19%	**9**%	14%	49%	49%	39%	**10**%	49%	67%	43%	19%	15%	36%	**34**%	**34**%	98%	61%	94%	**19**%	25%	27%	26%	54%	21%	28%	41%	**5**%	52%	56%	9%
Top-1 Value ↑	43.81%	**44.48**%	44.38%	39.07%	39.07%	41.13%	**46.08**%	42.62%	39.59%	41.93%	44.07%	45.05%	**44.85**%	44.65%	44.81%	36.61%	43.65%	38.79%	15.99%	15.81%	15.75%	15.99%	14.70%	**16.18**%	16.07%	15.66%	**17.01**%	15.28%	14.74%	16.73%
Top-5 (median) ↓	10%	**9**%	**9**%	35%	30%	19%	8%	36%	65%	37%	12%	**7**%	38%	**17**%	**17**%	95%	38%	73%	**10**%	12%	11%	29%	41%	12%	10%	27%	**8**%	35%	43%	24%
Top-5 (1^{st}-Q) ↓	9%	**7**%	9%	32%	25%	18%	6%	22%	63%	32%	11%	**5**%	33%	**14**%	**14**%	94%	34%	58%	9%	9%	**7**%	16%	39%	**7**%	7%	13%	**6**%	24%	38%	10%
Top-5 (3^{rd}-Q) ↓	11%	**9**%	10%	38%	31%	20%	**10**%	47%	72%	40%	15%	**8**%	40%	**18**%	23%	96%	47%	79%	**13**%	17%	15%	41%	46%	27%	11%	43%	**10**%	45%	49%	27%
Top-5 Value ↑	44.02%	**44.45**%	44.17%	40.49%	40.88%	43.22%	**44.87**%	42.39%	39.06%	41.48%	44.35%	44.78%	43.93%	**45.33**%	45.20%	38.78%	43.79%	42.08%	16.22%	16.11%	**16.31**%	15.13%	14.78%	15.82%	16.49%	15.39%	**16.56**%	14.95%	14.84%	15.80%
DNN	DenseNet100bc						PreResNet110						PreResNet20						VGG19						VGG16					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	8%	6%	9%	74%	74%	**4**%	**0**%	18%	22%	74%	24%	5%	**0**%	21%	30%	95%	15%	75%	14%	25%	**11**%	42%	57%	19%	21%	19%	**9**%	60%	35%	26%
Top-1 (1^{st}-Q) ↓	3%	3%	3%	66%	66%	**2**%	**0**%	10%	18%	66%	18%	1%	**0**%	18%	18%	95%	8%	64%	10%	19%	**3**%	19%	43%	13%	5%	**4**%	5%	49%	20%	18%
Top-1 (3^{rd}-Q) ↓	**10**%	**10**%	12%	79%	79%	64%	**4**%	24%	28%	81%	28%	8%	**2**%	48%	52%	97%	31%	79%	**23**%	36%	36%	68%	69%	36%	34%	69%	**12**%	81%	68%	34%
Top-1 Value ↑	**39.07**%	38.73%	38.69%	32.78%	32.78%	37.54%	**44.89**%	42.00%	40.68%	37.20%	41.39%	43.72%	**44.89**%	41.33%	40.91%	34.70%	42.55%	39.12%	18.74%	18.58%	**18.93**%	18.04%	17.77%	18.62%	17.55%	17.32%	17.85%	16.53%	17.12%	**17.57**%
Top-5 (median) ↓	**6**%	14%	11%	48%	39%	31%	**10**%	18%	40%	69%	14%	**10**%	**15**%	42%	41%	73%	27%	59%	**20**%	29%	26%	46%	52%	24%	22%	47%	**15**%	52%	38%	34%
Top-5 (1^{st}-Q) ↓	**5**%	13%	8%	43%	35%	28%	**6**%	11%	36%	65%	12%	9%	**14**%	35%	34%	67%	21%	54%	**12**%	20%	20%	36%	41%	19%	18%	25%	**11**%	42%	33%	28%
Top-5 (3^{rd}-Q) ↓	**10**%	16%	15%	52%	46%	34%	**12**%	24%	44%	74%	17%	**12**%	**23**%	45%	44%	75%	30%	75%	**30**%	37%	36%	50%	61%	40%	26%	51%	**24**%	60%	39%	37%
Top-5 Value ↑	**38.85**%	37.97%	38.45%	34.73%	34.75%	36.72%	**42.91**%	41.85%	39.40%	37.37%	42.62%	42.52%	**42.69**%	41.03%	41.17%	37.43%	42.03%	39.24%	**18.73**%	18.56%	18.55%	18.12%	18.01%	18.63%	17.62%	17.05%	**17.86**%	16.75%	17.34%	17.41%
DNN	ROBERTA						XLNet						BERT						DistilBERT						Electra					
Similarity ($\mathcal{P}$)	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}	*PWCCA*	*CKA*	*Ortho*	*Acc*	*Dis*	J_{Div}
Top-1 (median) ↓	**6**%	34%	64%	25%	11%	**6**%	30%	**5**%	10%	16%	**5**%	12%	20%	**6**%	19%	52%	14%	8%	10%	35%	**5**%	71%	22%	24%	**11**%	22%	30%	49%	19%	**11**%
Top-1 (1^{st}-Q) ↓	**3**%	28%	61%	13%	6%	**3**%	19%	**3**%	6%	11%	**3**%	8%	11%	**2**%	13%	24%	6%	3%	**2**%	24%	5%	15%	18%	8%	5%	15%	22%	46%	9%	**3**%
Top-1 (3^{rd}-Q) ↓	**9**%	50%	67%	53%	17%	**9**%	48%	**8**%	17%	22%	**8**%	15%	42%	**13**%	42%	83%	50%	26%	28%	40%	**11**%	77%	29%	30%	17%	31%	35%	56%	26%	**15**%
Top-1 Value ↑	22.01%	16.33%	14.55%	18.37%	21.42%	**22.08**%	19.22%	**23.61**%	22.51%	21.23%	**23.61**%	22.32%	29.80%	**31.59**%	29.49%	26.99%	29.67%	31.24%	**32.20**%	30.79%	32.59%	28.37%	31.39%	31.60%	24.51%	23.03%	22.40%	20.83%	23.42%	**24.93**%
Top-5 (median) ↓	20%	46%	54%	20%	**8**%	**8**%	52%	**8**%	12%	17%	10%	9%	21%	18%	24%	37%	22%	**15**%	19%	19%	**15**%	46%	16%	21%	**9**%	14%	16%	41%	14%	11%
Top-5 (1^{st}-Q) ↓	16%	43%	52%	13%	7%	**5**%	43%	8%	11%	13%	8%	**7**%	15%	12%	20%	30%	19%	**12**%	13%	15%	**10**%	44%	13%	16%	**6**%	12%	13%	40%	11%	8%
Top-5 (3^{rd}-Q) ↓	26%	48%	57%	44%	11%	**8**%	56%	**9**%	13%	23%	11%	10%	27%	**20**%	33%	42%	29%	**20**%	**21**%	23%	**21**%	48%	**21**%	26%	**15**%	17%	26%	50%	**15**%	16%
Top-5 Value ↑	19.12%	15.95%	15.08%	18.53%	21.37%	**21.46**%	18.20%	**22.63**%	21.65%	21.25%	22.44%	22.26%	30.03%	**30.29**%	29.51%	28.42%	29.82%	30.06%	**31.59**%	**31.59**%	31.77%	28.82%	31.52%	31.17%	**24.61**%	24.17%	23.86%	21.37%	23.82%	24.36%

Findings 3: Similarity metrics that showed good correlations generally enable the selection of good test sets for transferability. Even if said metrics are not necessarily the best for all DNN models under test, they will work better on average. Moreover, a selection of test sets based on the metric is always better than a random selection, further showing the utility of the method. Property coverage varies between DNN models under test/generation technique/property, but it's possible to cover up to $\sim$ 45% of the T_O property with a single test set.

As we have seen in previous RQs, the average $\mathcal{M}_{\mathcal{O}}$ coverage for each T_O of a DNN model under test for a given reference test set does not exceed 50%. Thus, even by transferring one good test set, the coverage is relatively low and can be improved. One straightforward way to improve the coverage would be to combine different test sets. Ideally, we would like the combined test sets to individually cover as much of a property (fault-type or neuron activation bands) as possible while individually covering different regions of the property to increase coverage.

Leveraging previous similarity metrics, we propose two heuristic criteria for combining test sets. First, since we have shown that some similarity metrics correlate with $\mathcal{M}_{\mathcal{O}}$ in the previous RQs, the n-th test sets of the most similar DNN models should generally cover more the property. If there is little overlap in the covered property (for instance, test sets not covering the same subset of fault types), the combination of said test sets should naturally improve the property coverage. We call this criterion *Overall Best First* (OBF) in the following. On the other hand, if the overlap in terms of covered property for the test sets is more pronounced or if the test sets individually do not cover much relatively of the property, the combination will not improve much in terms of the property. In that case, it might be better to instead combine test sets from different DNN model types as they can mitigate the overlap. In that case, we could combine test sets of the n-th most similar DNN model types. To do this, for each DNN model type (e.g., *RoBERTa*, *XLNet* etc.) we can determine the most similar (according to the chosen similarity) to the DNN model under test and use their combined test sets together. We call this criterion *Each Best First* (EBF) in the following.

As motivating examples to illustrate the defined criteria, we selected two DNN model seeds (one RoBERTa and one DenseNet100bc) on two different generation techniques (*Text*-based and *GAN*-based generation technique). For each of those DNN model seeds, we took available reference test sets and extracted the fault types on a given DNN model under test for each reference test set after using the clustering procedure defined in Section 6.3. We created vectors of k dimensions, where each dimension corresponds to one fault type identified for

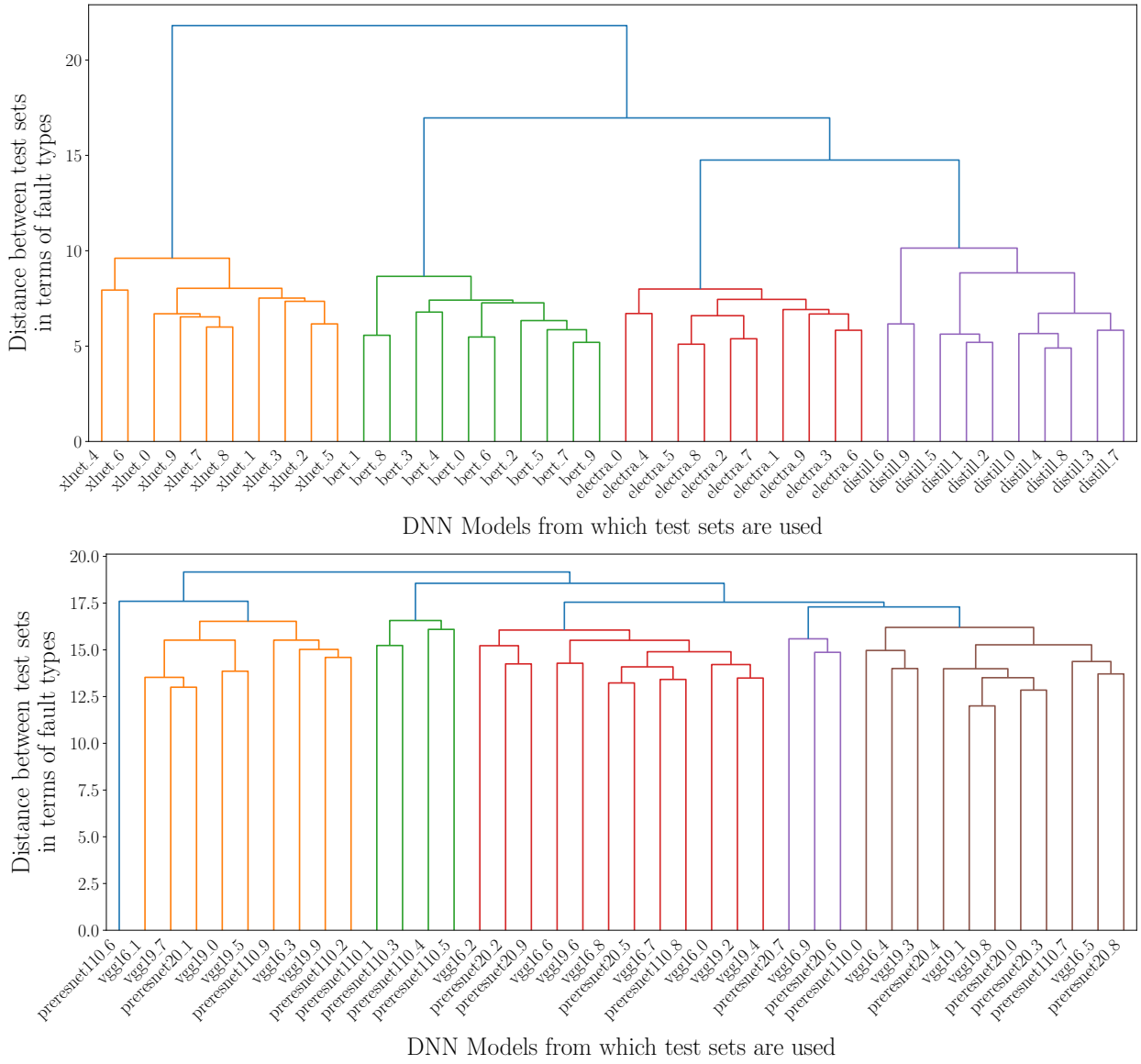


Figure 6.6 Distances between test sets for the fault types property on the DNN model under test. (**Top**) *Text*-based generation technique when DNN model under test is of type *RoBERTa*, (**Bottom**) *GAN*-based generation technique when DNN model under test is of type *DenseNet100bc*.

the T_O of the DNN model under test. The value of each dimension represents how many inputs of the reference test set were clustered in the given fault type cluster. We then leverage hierarchical clustering [227] to cluster the reference tests set for each DNN model under test according to their distribution across fault types. We can represent the results using a dendrogram for each DNN model under test. Results are presented in Figure 6.6.

The dendrogram represents, from the perspective of a DNN model under test, the distance

Table 6.7 Overall Best First, Each Best First and Random criterion score for the property based on fault-type coverage. Results are given for each generation technique and each DNN model under test using the chosen similarity metrics. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We highlight in bold the highest results.

Fuzz	DNN	DenseNet100bc			PreResNet110			PreResNet20			VGG19			VGG16		
	Fault types cov.	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	**47.81**	**46.61**	**49.02**	50.73	49.38	51.38	**52.46**	**51.71**	**52.96**	65.28	60.74	67.14	62.63	60.26	64.07
	OBF 2 ↑	**47.81**	**46.61**	**49.02**	**54.11**	**52.85**	**55.28**	49.91	47.69	51.52	**67.03**	**61.54**	**70.23**	**63.65**	**62.25**	**66.67**
	rand 2 ↑	46.19	43.02	49.48	50.26	47.74	52.68	50.21	47.80	52.24	58.63	54.94	62.03	55.60	51.98	59.13
	EBF 3 ↑	59.47	58.56	60.62	63.40	61.16	64.47	**66.15**	**64.40**	**66.79**	74.07	71.34	76.22	71.93	70.93	73.07
	OBF 3 ↑	**62.31**	**61.73**	**64.49**	**65.17**	**63.84**	**66.04**	63.12	60.41	65.47	**77.75**	**71.93**	**78.87**	**75.55**	**73.63**	**77.30**
	rand 3 ↑	58.06	54.16	61.91	62.05	59.59	64.50	62.01	60.06	64.08	70.38	67.57	73.06	68.60	65.61	71.31
	EBF 4 ↑	67.57	66.63	68.10	71.15	**70.28**	72.45	**73.12**	**71.62**	74.27	79.68	78.83	80.98	77.11	76.53	79.02
	OBF 4 ↑	**71.80**	**71.21**	**73.37**	71.69	69.40	72.30	71.35	68.29	**74.62**	**82.75**	**79.38**	**83.76**	**81.07**	**78.17**	**82.69**
	rand 4 ↑	68.97	66.49	71.71	**71.87**	69.48	**74.07**	71.20	69.37	73.50	78.86	76.00	80.89	76.92	74.37	79.10
GAN	DNN	DenseNet100bc			PreResNet110			PreResNet20			VGG19			VGG16		
	Similarity	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	40.67	38.87	42.58	39.85	37.42	40.90	**38.13**	**36.45**	**40.04**	54.94	53.12	55.95	53.31	50.80	54.46
	OBF 2 ↑	**44.64**	**41.55**	**45.75**	**40.98**	**40.26**	**42.67**	37.69	36.30	39.95	**55.56**	**54.55**	**56.85**	**53.80**	**51.83**	**54.92**
	rand 2 ↑	38.85	37.11	40.70	35.55	33.93	37.37	35.69	34.23	37.70	51.98	50.45	54.28	51.44	49.11	53.27
	EBF 3 ↑	52.72	49.74	54.67	49.42	47.33	50.64	**50.55**	47.78	**52.69**	67.07	66.26	67.74	65.45	63.54	66.88
	OBF 3 ↑	**57.41**	**56.61**	**58.91**	**52.79**	**51.23**	**53.18**	49.60	**47.92**	51.45	**67.23**	**66.42**	**68.40**	**65.92**	**64.76**	**67.60**
	rand 3 ↑	50.63	48.80	52.47	46.68	44.71	48.99	47.80	46.20	49.51	64.76	63.13	66.87	63.80	62.17	65.50
	EBF 4 ↑	59.95	57.83	61.77	57.89	55.55	59.36	**58.86**	56.65	**61.54**	74.97	73.11	75.84	73.11	71.96	73.76
	OBF 4 ↑	**65.72**	**64.39**	**66.73**	**61.94**	**61.49**	**63.82**	58.12	**57.84**	59.69	**76.03**	**74.92**	**77.40**	**75.69**	**75.04**	**76.35**
	rand 4 ↑	60.24	58.96	62.19	57.43	55.18	59.55	57.11	55.24	58.94	73.77	71.79	75.43	72.69	70.83	74.70
Text	DNN	ROBERTA			XLNet			BERT			DistilBERT			Electra		
	Similarity	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	**53.15**	**50.82**	**56.21**	**52.22**	**49.77**	**53.08**	**53.58**	**51.68**	**56.61**	**59.63**	**57.26**	**62.50**	**54.34**	**53.52**	**56.91**
	OBF 2 ↑	49.30	48.82	52.54	48.59	47.15	49.66	52.55	49.30	53.67	50.59	50.05	51.76	51.22	49.92	53.08
	rand 2 ↑	42.28	37.01	49.30	43.59	39.22	47.78	43.66	38.45	50.98	45.75	36.11	51.39	44.63	39.77	49.77
	EBF 3 ↑	**64.60**	**61.72**	**66.14**	**65.85**	**63.86**	**67.07**	**65.65**	**62.30**	**66.67**	**69.43**	**64.54**	**71.01**	**67.01**	**66.38**	**68.09**
	OBF 3 ↑	59.77	56.75	61.32	56.51	54.93	57.29	59.88	57.84	61.87	56.00	54.06	60.09	60.94	58.40	63.30
	rand 3 ↑	54.30	48.83	60.69	53.85	49.92	59.50	55.92	50.18	62.96	57.54	53.77	63.41	59.77	56.24	63.85
	EBF 4 ↑	**73.91**	**72.40**	**76.06**	**76.80**	**73.69**	**78.37**	**75.11**	**72.60**	**76.11**	**76.31**	**73.82**	**78.26**	**75.74**	**73.20**	**77.13**
	OBF 4 ↑	64.85	63.15	66.83	61.67	60.52	62.79	66.67	63.30	67.34	61.37	59.13	64.01	67.57	66.32	70.26
	rand 4 ↑	69.12	64.02	72.09	68.26	63.97	71.79	69.94	64.57	73.24	70.11	66.10	74.29	70.18	66.05	73.06

between each test set in terms of fault types. Remark that however, it does not mean that those test sets cover more or less fault types than the other test sets on the DNN model under test. This simply quantifies the distance between test sets. So, for instance, the top dendrogram can be read as: "from the point of view of the given *RoBERTa* DNN model under test, the reference test sets from *XLNet* (DNN model seed 4) and *XLNet* (DNN model seed 6) have a difference in terms of fault types of 8". In the *Text*-based generation technique, one can see that reference test sets for a given DNN model types (e.g., *XLnet*, *BERT* etc.) are closer in terms of fault types compared to test sets of other DNN model types. So they would lead to more similar fault types from the point of the DNN model under test *RoBERTa*. In that case, using *EBF* as the selection criteria is the sound choice to improve coverage. On the other hand, in the *GAN*-based generation technique, reference test set clusters are not reliant on the DNN model types. In that case, one should rely on the absolute coverage of

Table 6.8 Overall Best First, Each Best First and Random criterion score for the property based on neuron coverage. Results are given for each generation technique and each DNN model under test using the chosen similarity metrics. For the same DNN model type (e.g., *DenseNet100bc*), the results are given for all DNN model seeds. We highlight in bold the highest results.

Fuzz	DNN	DenseNet100bc			PreResNet110			PreResNet20			VGG19			VGG16		
	Fault types cov.	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	60.87	58.35	61.77	60.88	60.31	62.19	**64.85**	61.72	**65.54**	21.81	20.42	23.63	23.90	20.24	28.06
	OBF 2 ↑	**63.23**	**63.11**	**63.59**	**63.72**	**62.93**	**64.15**	63.57	**62.07**	64.02	**22.84**	**21.28**	**25.54**	**25.49**	**21.45**	**30.04**
	rand 2 ↑	57.50	55.45	59.70	58.75	57.16	60.34	62.36	60.33	63.87	20.52	18.26	23.14	22.09	18.37	26.03
	EBF 3 ↑	70.78	68.85	71.28	70.31	69.77	71.49	**73.64**	**71.64**	**74.11**	26.94	24.96	30.18	30.38	24.94	35.32
	OBF 3 ↑	**73.46**	**73.04**	**73.72**	**72.59**	**72.28**	**73.08**	72.69	71.62	73.11	**27.64**	**26.41**	**33.20**	**32.73**	**26.96**	**38.35**
	rand 3 ↑	68.77	66.95	70.23	68.94	67.84	69.85	72.15	70.66	73.31	25.46	23.37	29.10	28.92	23.52	34.39
	EBF 4 ↑	76.86	75.32	76.95	76.41	75.33	77.13	**78.92**	**77.30**	**79.18**	29.76	28.38	33.87	34.53	27.75	39.85
	OBF 4 ↑	**79.12**	**78.83**	**79.54**	**78.21**	**77.64**	**78.66**	78.29	77.26	78.75	**32.55**	**29.72**	**37.20**	**38.74**	**30.89**	**45.04**
	rand 4 ↑	76.00	75.00	77.16	75.63	74.67	76.48	77.90	76.67	78.87	29.47	27.27	33.95	34.33	27.42	40.52
GAN	DNN	DenseNet100bc			PreResNet110			PreResNet20			VGG19			VGG16		
	Similarity	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	55.35	54.62	56.02	60.99	60.69	61.48	**62.82**	61.57	**63.70**	**26.33**	23.98	29.21	28.09	24.08	32.14
	OBF 2 ↑	**58.63**	**57.27**	**59.70**	**62.51**	**62.24**	**63.35**	62.70	**62.32**	63.44	26.31	**24.78**	**29.34**	**28.97**	**24.14**	**32.16**
	rand 2 ↑	53.52	51.67	54.84	57.12	55.73	58.51	59.62	58.38	60.83	25.41	23.33	28.29	27.74	22.12	30.29
	EBF 3 ↑	65.37	64.99	65.95	69.69	69.01	70.31	71.76	71.19	72.66	31.91	29.52	35.91	34.95	29.18	38.77
	OBF 3 ↑	**69.60**	**68.49**	**70.06**	**71.52**	**71.14**	**72.17**	**72.41**	**71.63**	**73.12**	**32.79**	**30.25**	**37.22**	**35.85**	**30.15**	**40.74**
	rand 3 ↑	64.40	63.18	65.53	67.11	65.98	68.28	69.90	68.89	71.05	31.38	29.13	35.95	34.60	27.90	38.14
	EBF 4 ↑	71.41	70.99	72.24	74.72	74.06	75.08	77.19	76.84	77.57	37.14	32.91	41.70	40.03	33.42	44.94
	OBF 4 ↑	**76.01**	**75.23**	**76.19**	**77.31**	**76.78**	**77.46**	**78.22**	**77.24**	**78.39**	**38.95**	**34.41**	**43.39**	**41.93**	**35.17**	**46.25**
	rand 4 ↑	72.16	71.32	72.90	74.38	73.56	75.32	76.12	75.14	76.97	36.81	33.07	42.43	40.85	32.17	44.60
Text	DNN	ROBERTA			XLNet			BERT			DistilBERT			Electra		
	Similarity	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q	Median	1^{st}-Q	3^{rd}-Q
Selection Heuristic	EBF 2 ↑	**33.57**	30.98	34.19	35.37	34.26	35.89	**42.25**	**33.74**	**46.96**	**50.50**	**48.60**	**52.14**	37.07	36.49	**41.29**
	OBF 2 ↑	33.06	**31.72**	**34.70**	**37.38**	**36.84**	**37.66**	41.90	33.15	46.00	47.65	45.46	49.86	**37.37**	**36.77**	39.54
	rand 2 ↑	27.42	24.85	31.54	31.60	29.27	33.96	38.84	27.42	45.26	46.56	43.50	49.10	35.04	33.04	37.71
	EBF 3 ↑	40.50	39.29	41.93	44.87	44.28	45.66	**52.65**	40.75	**57.88**	**60.23**	**59.54**	**61.58**	**47.28**	**46.86**	**50.64**
	OBF 3 ↑	**42.17**	**40.61**	**43.94**	**46.32**	**44.73**	**47.47**	50.62	**42.20**	54.97	56.51	54.37	59.42	46.36	45.24	48.90
	rand 3 ↑	35.96	33.23	40.48	40.68	38.33	43.81	49.37	35.96	56.90	59.37	57.11	61.42	46.33	44.48	48.98
	EBF 4 ↑	47.50	46.38	49.33	53.17	51.98	53.66	**61.34**	**47.86**	**67.14**	**68.45**	**67.57**	**69.75**	**55.96**	**55.40**	**59.18**
	OBF 4 ↑	**48.62**	**47.67**	**50.69**	**53.50**	**52.57**	**53.84**	57.95	48.66	61.78	63.45	61.33	66.20	53.47	51.97	57.18
	rand 4 ↑	46.72	43.54	49.01	51.19	47.57	53.36	58.26	46.72	65.48	67.78	65.78	69.40	54.58	52.72	56.12

each of the test sets for the combination to maximize the coverage, i.e., the *OBF* criteria.

To measure the effectiveness of our criteria, for each generation technique, we evaluate both OBF and EBF criteria on each DNN model under test. We do so by taking the k-th best DNN models in terms of each criterion following the ranking of the chosen similarity. We varied k from 2 to 4 to see if the coverage increased or stagnated as we added more test sets. To put it in perspective, we also compare the criteria to randomly sampling the same number of reference test sets. To do so, we varied k from 2 to 4 and sampled 30 test sets randomly for each k. For the choice of similarity, we picked $PWCCA$ and J_{Div} as those similarities worked reasonably well both in terms of correlations and $Top-1/Top-5$ metrics in previous research questions. Results are presented in Table 6.7 for the property based on Fault type coverage and in Table 6.8 for the property based on neuron coverage.

Results show that, for the chosen similarities, with a fixed generation technique and property, there is a criterion better than the other across most DNN models under test. For both neuron and fault-type coverage properties, *EBF* is better for the *Text*-based generation technique and *OBF* is better for *GAN*-based and *Fuzz*-based generation techniques. The reason why one criterion works best on a particular generation technique is as shown with the dendrogram examples: depending on the DNN model under test, certain reference tests have more overlapping faults while belonging to the same DNN model type. For the fault-type property, coverage reaches ∼60-80% of the fault-type of T_O depending on the DNN model type and generation techniques. For the neuron property, coverage reached ∼40-80% but has much more variation across generation techniques and DNN model types. Notably, while more than doubling, coverage reaches only ∼40% on *VGG* DNN model types and ∼50% on *RoBERTa* and *XLNet*. More importantly, one can see that randomly sampling test sets is not a good approach as it rarely leads to more coverage compared to using the proposed heuristics criterion and the similarity. Thus, selecting test sets based on DNN model similarity can bring benefits in increasing the coverage when transferring test sets.

Findings 4: At fixed similarity, generation technique and property, one heuristic selection criterion for test sets combination is best overall. In our case, *EBF* works better on *Text*-based generation technique while *OBF* works better on *Fuzz*-based and *GAN*-based generation techniques. Moreover, not only does combining test sets using ranks of similarity help in increasing property coverage, but it does so better than randomly sampling them. Thus, carefully selecting test sets leveraging defined similarity is beneficial for transferring test sets.

6.5.5 RQc_4: Is transfer with GIST more effective than generating test cases from scratch from a trade-off property covered/execution time point of view

To demonstrate the effectiveness of GIST, we first measure the execution time of each step of the process (*Offline* and *Online*) as well as the range of execution time of each of the test case generation procedures in our experiments. Results are presented in Table 6.9. We used a computer with the following configuration to compute the execution time of experiments: Ubuntu 22.04, GPU (8 GB VRAM), 16 Go RAM and CPU with 8 cores.

We observe that the different phases have different impacts on the overall process: feature extraction is proportional to the DNN model's size and so larger DNN models such as is the case in the text base dataset (i.e., RoBERTa, XLNet etc.) will take a longer time to

Generation technique		Offline					Online
Type		Features	Property (Fault / neuron)		Correlation (Fault / neuron)		Selection
Fuzz	∼ 2,000 - 8,000	∼ 1,100	∼ 7,000	∼ 6	∼ 1,700	∼ 2,000	∼ 1-20
GAN	∼ 3,000 - 5,000	∼ 1,100	∼ 8,800	∼ 6	∼ 1,800	∼ 2,000	∼ 1-20
Text	∼ 12,000 - 14,000	∼ 4,500	∼ 6,265	∼ 10	∼ 2,000	∼ 2,500	∼ 1-30

Table 6.9 Execution time of each part of *GIST* (*Offline* and *Online*) in seconds. We detail each part of the *Offline* procedure and give the execution time of the test generation technique as a comparison.

run. On the contrary, the correlation computation is proportional to the number of comparisons to do and the number of similarities to use. For the similarities, the execution time varies a lot depending on which one is used: representational similarities will take a longer time than functional ones. For instance, functional similarities take on average 1-2 seconds while similarities such as PWCCA can take up to 30 seconds per DNN model under test as highlighted in the *Online* part. For those representational metrics, we used a PyTorch version of the implementation as provided by Ding et al. [208] to facilitate replication. Lastly, the execution time on the property drastically varies: while the fault type property using UMAP/HDBSCAN takes around 7,000 sec, the neuron coverage-based one is much less expensive to compute (around 6 seconds). As such, the property chosen can have a high impact on the execution time of GIST. Note that for the fault type, we relied on the same kind of libraries and procedures used by Aghababaeyan et al. [170]. So, for instance, the UMAP/HDBSCAN used in our experiments is the original CPU-only implementation[2] to allow for replication, which impacts execution time. However, GPU-based implementations such as the one in the cuML library [228] could drastically improve execution time.

In the *Generation technique* column, we show the average execution time range for running the generation technique. Execution time varies depending on the DNN model and the technique. For instance, for the fuzzing method applied on a smaller less robust DNN model such as *PreResnet20* will take on average around 2,300 seconds to obtain the 1,000 generated inputs requested. On the contrary, applying on a more robust and bigger DNN model such as *PreResnet110* will take on average around 8,000 seconds. Finally, the selection of relevant test sets in the *Online* part has a very small cost to compute the most similar DNN model to the DNN model under test given the similarity (1-2 seconds for functional similarity and up to 30 seconds for representational). The follow-up inference cost is dependent on the number of inputs being transferred and the DNN model size but will inevitably be smaller

[2] `https://umap-learn.readthedocs.io/en/latest/` and `https://hdbscan.readthedocs.io/en/latest/`

than the *Generation* cost which generally requires additional computation (such as gradient backpropagation, multiple inference passes etc.).

Once we have measured the execution time, we can define an index r to measure the trade-offs of the transferability (*Offline* + *Online*) against reapplying the generation techniques. This r is inspired by existing performance metrics [229]. We randomly sample 100 execution times for different DNN model seeds and collect their property coverage obtained in RQc_3 (through EBF/OBF criteria, with 4 test sets). We equivalently get the execution time taken by the generation of tests for this DNN model seed and the execution time it would take to apply *GIST* using data from Table 6.9. We then obtained the index r by dividing the property coverage obtained by applying GIST and a time ratio t. This time ratio t is itself obtained by dividing the execution time of applying GIST for transferring n times and the execution time of applying the test cases generation technique on n DNN model under test. The rationale is to show what is gained (in terms of property coverage) compared to what is lost (in terms of execution time) against just applying the traditional generation technique (ratio *Offline*/*Online* and generation technique execution time). For instance, $r = 1$ means that the property coverage was 100% and the *Offline*/*Online* execution took as much time as applying the generation techniques on n DNN models under test. It can also mean that the property coverage was 50% but the *Offline*/*Online* execution took half the time of the generation techniques on n DNN models under test. Generally, a r value above 1 means that the transfer is more efficient and a r value below 1 means that it's less efficient than reapplying the generation techniques on the n DNN models under test. Results are given for the fuzzing generation technique in Figure 6.7, for the GAN generation technique in Figure 6.8 and for the text-based generation technique in Figure 6.9.

For the fuzzing generation technique, in scenario (a) (transferring only on one DNN model), *fault-types* based coverage property transfer is not more efficient than reapplying the generation technique. However, *neuron* based coverage property is already more efficient (median $r \sim 1.3$), even though there is a high variability depending on the DNN models under test. The further we increase the number of generation techniques required (i.e., if we had to apply the technique on multiple DNN models under test), the more efficient the transfer becomes as the *Offline* cost is paid once with only subsequent cheap *Online* execution. For four DNN models under test, both property transfers are more efficient than actually applying the generation techniques from scratch on all of them (scenario (c)). For just two DNN models under test, if the execution time of applying the test cases generation technique is high ($\sim$ 8,000 s such as a *DenseNet100bc* or *PreResNet110* DNN model types), then both property transfers are already more efficient (scenario (d)). We obtain similar results for the text-based generation technique, with the transfer already being more efficient starting from two DNN

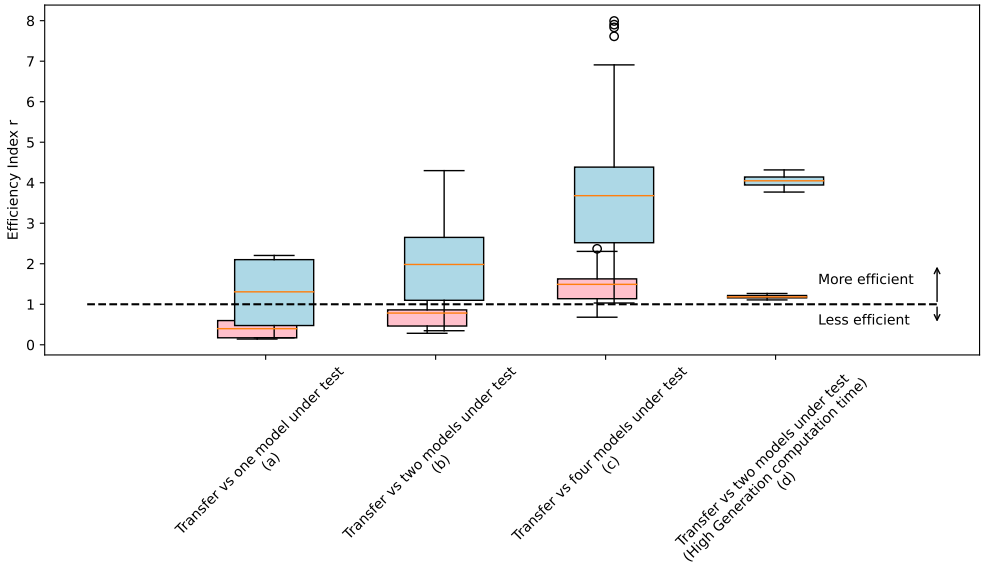


Figure 6.7 Efficiency index r for different comparison scenarios of applying GIST vs applying the generation techniques (Fuzzing) on n DNN models under test. Red boxplots are results for *fault-type* based coverage, blue boxplots are results for *neuron* based coverage.

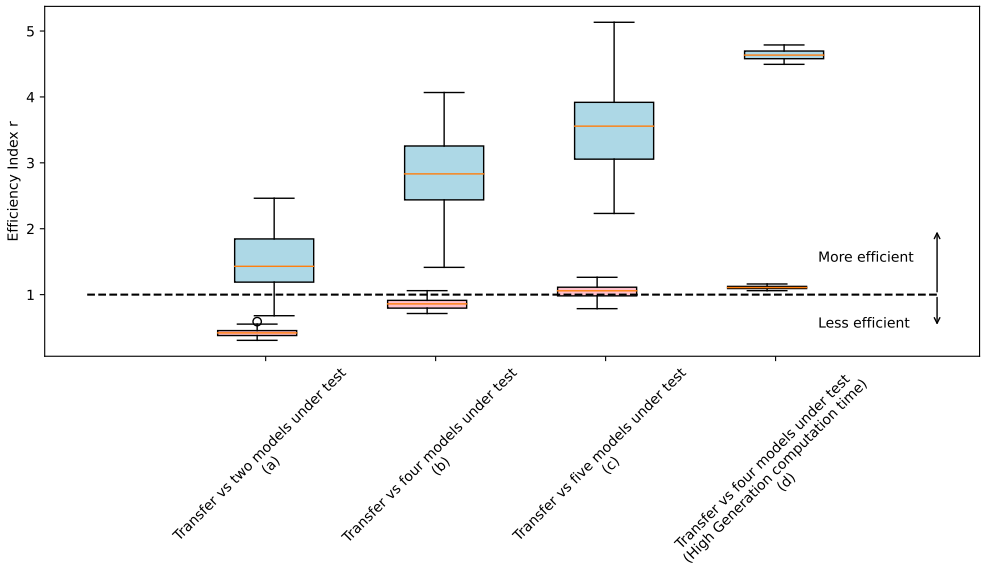


Figure 6.8 Efficiency index r for different comparison scenarios of applying GIST vs applying the generation techniques (GAN) on n DNN models under test. Red boxplots are results for *fault-type* based coverage, blue boxplots are results for *neuron* based coverage.

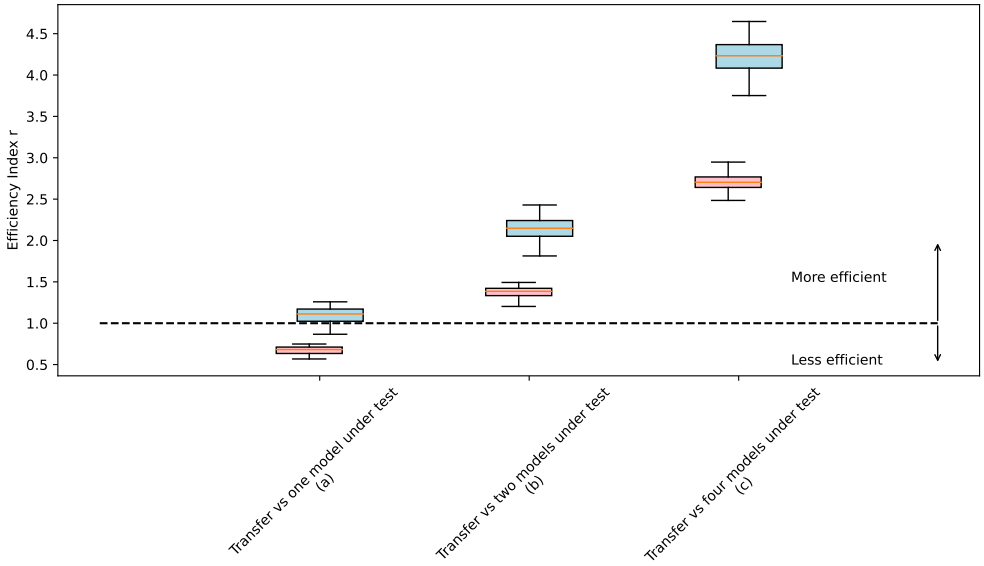


Figure 6.9 Efficiency index r for different comparison scenarios of applying GIST vs applying the generation techniques (*Text*-based) on n DNN models under test. Red boxplots are results for *fault-type* based coverage, blue boxplots are results for *neuron* based coverage.

models under test. The results for the GAN generation technique follow a similar pattern, even though it does require a higher number of DNN models under test to transfer on for the fault types property based transfer to become efficient. The results for the neuron-based property are not affected. We explain the difference because of the added execution time in calculating the fault type clusters due to different hyperparameters. As we explained at the beginning of this section, the execution time for the fault type property is CPU-based only due to the libraries used. And so, even in that case, the transfer shows some benefit that would likely be improved by using more optimized calculations. Note that r accounts both for coverage and execution time. If only execution time is of importance, transferring can already be more efficient with a lower number of generation techniques applied.

Findings 5: While the upfront cost of GIST (*Offline* part) can be high, it is paid only once. Thus, GIST transfer becomes profitable as more test case generation techniques (and so the DNN model under test) are required. This is mainly because the subsequent *Online* parts are fast. In particular, it takes two to four reapplication of the generation techniques on DNN models under test for the transfer to be effective in terms of property covered / execution time ratio. In terms of execution time only, transferring can be even faster compared to using a generation technique from scratch on a DNN model under test.

6.6 Discussion

The success of the envisioned transfer using GIST is dependant on both $\mathcal{P}_O$ and $\mathcal{P}$. While we investigated transferability in terms of fault types and neuron coverage, $\mathcal{P}_O$ could be any property we would want to transfer. The first requirement would be to find a proxy $\mathcal{P}$ that only relies on data available during the online phase, i.e., DNN models' weights or train data for instance, but not the actual test sets. The second requirement would be to show that there exists a correlation and that some empirical criterion can be leveraged to transfer a test set. Good candidates for $\mathcal{P}$, are the representational/functional similarities metrics we used in our experiments. While other proxies $\mathcal{P}$ could be explored, we believe those similarity metrics should be checked first.

In practice, GIST can be used as shown in Figure 6.2: after investigating the correlation between $\mathcal{P}_O$ and all $\mathcal{P}$ available on the benchmark of available DNN models and test sets, one determines which $\mathcal{P}$ is best. This is the offline phase, similar to what we did in our experiments. At the online phase, a new DNN model under test is trained without having access to its test set through the generation technique $\mathbf{G}$. The user then calculates the similarity with the chosen metric between the DNN model under test and all benchmark DNN models. Then, the test sets being determined the best according to a defined criterion are transferred to the DNN model under test and we have empirical evidence that the chosen test sets are on average better than other available test sets in terms of the metric $\mathcal{P}_O$. That is to say, the chosen test sets can effectively cover similar properties as to what we would have obtained by generating the test set from scratch. There are however some trade-offs within the framework that one must keep in mind and that we discuss in the following:

6.6.1 Time complexity

While in the *Online* phase, not much information is needed on the DNN model under test, there is some extensive computation being made on the benchmark during the *Offline* phase. As such, when it comes to trade-offs, *GIST* would shine in several cases:

1) If the *Offline* process has already been computed, similar to having access to a DNN model to transfer in transfer learning, then the cost of the transfer procedure is solely reliant on the *Online* part which is drastically inferior to the generation technique used.

2) If the *Offline* part hasn't been computed, then it should translate to planning on using the transfer for multiple DNN models. Indeed, while the upfront cost can be similar or more expensive than applying the generation technique once, it is paid only once. As such, any further application of the generation technique on a new DNN model seed or a new architecture will require new computation whereas *GIST* only requires a one-time computation. This is what we demonstrated in RQc_4. In particular, note that the *Offline* part can be done anytime as it does not require the DNN model under test contrary to the *Online*/generation process. This is analogous to how one would likely not train a DNN model for transfer learning if there is only one task on which the DNN model will be used.

3) Alternatively, still in the case of the *Offline* part not being computed, the transfer procedure could be used (even if it's not the intended usage) on one DNN model when the criteria cost is reduced. Indeed, while the fault-type based property might not be the most cost-effective for one DNN model under test, using the neuron coverage-based (or similar) property drastically reduces the cost and makes the process less costly than applying the generation technique, even for one DNN model under test. As such, GIST might still be useful even when the user only has one DNN model, but only for certain properties.

Finally, while one could certainly run all test sets available without selection, this incurs an inference cost proportional to the number of inputs and the number of test sets. Since we saw in RQc_3 that carefully selecting the test sets yields good results which are better than randomly selecting them, leveraging ranking similarity as GIST is a valid option. This is especially true when inference cost becomes prohibitive and because *Online* execution time is very low. Moreover, with the aim of potentially improving the DNN model by retraining with the test sets to alleviate faulty inputs, fewer inputs will lead to a lower training time.

6.6.2 Number of DNN model seeds and DNN model types

While the transfer paradigm could be applied in any situation, *GIST* requires a certain number of DNN model seeds/types to validate the proxy. This can limit its usage in situations where such DNN model seeds/types are harder to come by. Thus, aiming to reduce the number of DNN model seeds and/or types needed to select a similarity could help in making GIST more practical in those specific DNN model-limited tasks. However, this limitation can be mitigated by choosing smaller/simpler DNN models to build up the available DNN models and apply them to much more complex DNN models. All the publicly available DNN models and test sets of our replication package [217] can constitute a good starting point to further expand the available DNN models. As an example of this, we trained a *PreResnet1202* which is roughly $\sim$10$\times$ the computational complexity compared to $PreResnet110$ [54]. Applying the fuzzing generation technique on this DNN model took $40,000$ seconds to generate a test set of $1,000$ inputs following the same technique as we did in our experiments. Leveraging GIST, we could select appropriate test sets for transfer. In practice, this would for instance result in more than 70% of both properties being covered for the hypothetical T_O of this DNN model using the EBF/OBF criteria with 4 test sets transferred. As such, we can cover a reasonable amount of the property while saving a large amount of time.

6.6.3 Potential improvements venues

Having shown GIST's setup theoretically and its practical effectiveness, there are still several improvements that can be designed in future works. The main goal of our study was to introduce the idea that transferring test sets could be another avenue for DNN model testing. To show our approach worked, we focused on two properties (neuron and fault-types based), two modalities (image and text) along with test cases generation methods as well as several DNN model types (VGG, DenseNet or ResNet-like for images and BERT-like transformers for text) and showed that GIST could be leveraged to transfer test set. Nevertheless, future works could expand on other modalities, DNN models and properties. In particular, as we mentioned at the beginning of this section, this framework is general in the property to be transferred. While we only studied two properties, it would be interesting to compare the same DNN models and test sets on other properties, either using the similarity metrics as a proxy $\mathcal{P}$ or coming up with a new one. Secondly, on the topic of similarity metrics, it could be worthwhile to investigate other types of metrics we did not study here. The ones we chose are good samples of available ones, yet some such as Model Stitching [230] (which requires additional computation and information from the DNN models compared to the metrics we used) could also show good performance. Moreover, it could even be possible to leverage Deep

Metric Learning [231] to come up with a specific metric for a given property. This however would need to account for the constraint that in the online phase, i.e., we do not have access to the test set that would have been generated on the DNN model under test. Then, while *EBF* and *OBF* criteria are simple and straightforward heuristics to leverage similarities to combine test sets and improve coverage, they increase the number of test inputs. When inference computation is inexpensive, or even in our case where individual test sets do not have more than 1,000 individual inputs, this might not be a problem. However, this can become one for larger DNN models or even when using larger test sets. Moreover, combining test sets as a block will inevitably lead to multiple inputs covering the same fault type making them redundant. As such, leveraging test seed selection methods [167, 232, 233] on top of the similarity metrics could allow the sampling of an optimal test subset for this property, i.e., each input would for instance only cover one fault-type or neuron band. However, just as for the choice of the metric, this needs to take into account the constraint that, during the online phase, we do not have access to the test set of the DNN model under test. Finally, another future work could consider how to include the transfer property as an objective in the generation technique. In that sense, transferred test sets could serve as a basis of easily accessible test inputs and the generation technique could aim to complete this basis with additional test inputs. Those additional test inputs would cover the remaining part of the property not covered by the original basis. This could be a middle point between purely transferring test sets and plainly generating test inputs via the generation technique.

6.7 Threats to Validity

Construct validity: We provided a general framework with proper constraints and a clear frame: to find $\mathcal{P}_O$ and $\mathcal{P}$ with a clear correlation relation between them. This approach could be generalized to any case in which the above problem can be defined, independently of the $\mathcal{P}_O$ and $\mathcal{P}$. To establish such a relation, we used Kendall-*tau* as prescribed in the literature. We based our property $\mathcal{P}_O$ on existing metrics: either fault-type or neuron coverage-based. Each of those metrics has its existing limitations. However, the point of the framework is to show that a relation can be leveraged independently of the $\mathcal{P}_O$. We chose those metrics that were used in the literature. For the proxy, we used several similarities defined in the literature. As we can not know in advance what proxy can work for the correlation, several similarity metrics need to be used, which can be a threat to validity. Nonetheless, with the diverse pool we selected, we showed that a correlation exists with both our properties (fault types/neuron coverage). Finally, we showed that it could be leveraged to be of practical use. The comparison against random samples could be a threat to validity. We used 30

independent random samples as it is common in the literature [234]. Therefore, we believe our empirical evaluations validate and motivate GIST.

Internal validity: There are several internal validity in our study, namely the clustering approach for the fault types, the choice of DNN models, the test inputs data as well as the impact of the seed over the result. To mitigate all of them, we applied several strategies: for the clustering, we followed similar approaches leveraging dimensionality reduction before clustering and verified that clusters represent different fault types following previous papers methodology [170,235]. For the choice of DNN models, we picked different DNN model types that, while reaching similar accuracy on the dataset, have some differences and resemblances to investigate how our results would be affected. For the test input generation, we used different existing approaches. We made sure to use the same original test inputs for the generation to reduce potential bias when comparing DNN models. Finally, we made sure to use different DNN model seeds, to compare how our results would differ, limiting potential seed bias, and evaluating the robustness of the method against this effect.

External validity: To further the generalization of our study, we made sure to experiment using two datasets of different modalities as well as three different test input generation techniques. For each of those techniques, we leveraged five different DNN model types with ten DNN model seeds, each used to generate a test set. In all the cases, we showed that one similarity could be chosen which worked reasonably well across all DNN model seeds of all DNN models under tests. As such, we expect our results to generalize to other modalities, datasets, DNN models and test input generation techniques. Finally, we studied the transferability of two properties (fault types and neuron coverage), which further comforted us that the problem setting of GIST is sufficiently general that it could be leveraged in other situations.

Reliability validity: To increase the reliability of our study, we detailed our methodology and provided a replication package [217] with our artifacts to allow for replication of our experiments.

6.8 Chapter Summary

This chapter introduced GIST, a framework for test sets transferability in DL. The framework's goal was to tackle the cost of applying testing techniques for model-level faults in DNNs, which can induce a high computation time because of the required test inputs. By proposing to transfer the test sets with GIST, we can effectively reduce the cost of the testing techniques. That is, the test sets transferred to cover the property targeted (e.g., faults

detected) proportionally more than the ratio of time between applying the testing technique from scratch and applying GIST. Thus, we enable a better usage of those testing techniques despite the time constraint. Notably, using GIST for transfer is better than doing it randomly in all cases, and we can cover up to 80% of the properties targeted with the transferred test sets. Moreover, transfer starts becoming more effective than generating test sets from scratch starting at two DNNs when applying GIST. This transfer was based on leveraging the relation between DNNs through similarity metrics. Indeed, DNNs "learn" in a comparable way, so their test sets should present some likeness in terms of property, such as faults detected, which we exploited for the transfer. Moreover, transferred test sets could additionally serve as a basis for amplification to generate more test inputs at a reduced cost.

CHAPTER 7 DIFFEVAL: ASSESSING DIFFICULTY OF CODE GENERATION TASKS FOR LARGE LANGUAGE MODELS

While general metrics computed over benchmarks paint a macroscopic view of the benchmarks and of the LLMs' capacity, it is unclear how each task in these benchmarks assesses the capabilities of the LLMs. The objective of this chapter is to focus on a more local analysis through the lens of programming task *difficulty* for LLMs, aiming to identify the level of difficulty of the tasks to sort between easy and hard tasks. We generate diverse ways of formulating a task by leveraging transformations over the prompts. LLMs then assess those prompts, and their output is evaluated through code-related metrics. This leads to the definition of difficulty score per task, allowing us to identify and analyze hard tasks. Using two code generation benchmarks, we show that we can reliably identify the hard tasks within those benchmarks, highlighting that 17% and 24% of the tasks are hard for LLMs in both benchmarks. Through our analysis of task difficulty, we also characterize 6 practical hard task topics per benchmark, which we used to generate 15 new hard tasks for each benchmark.

7.1 Introdudction

After dealing with model-level faults for a particular DNN under test through the cost of testing techniques, in this chapter, we will tackle model-level faults through the lens of the effect on those faults of global evaluation, such as Accuracy, specifically in LLMs for code. Generally, to assess LLMs on code generation tasks and compare them to each other, benchmarks (i.e., test sets) are used. Such benchmarks comprise several *task* generally represented by a prompt. This way, the code generation capability of LLMs on programming tasks is assessed indirectly at the benchmark level based on general characteristics such as dependency level (e.g., function level being less difficult than class level) and through global metrics over the benchmarks. Moreover, the assessment of those tasks is conducted via a single input text called a prompt, a single formulation of the task, where the prompt formulation has been shown to impact the output of LLMs. This, however, introduces limitations as current benchmarks' usage and assessment paint a general yet incomplete picture, and they fail to provide a more fine-grained analysis of individual tasks within a benchmark. The framework proposed in this chapter aims for fine-grain analysis by identifying the level of difficulty of tasks in a benchmark to sort between easy and hard tasks. Identifying such difficult tasks could foster improvement and resilience of LLMs, as such hard tasks constitute model-level faults that are quite problematic as LLMs struggle to solve them properly no matter the

formulation of the prompt given.

In this study, we present *DiffEval*, a framework for assessing task difficulty for LLMs and crafting new tasks based on identified hard tasks. DiffEval is orthogonal to the existing benchmark's usage and works on top of available benchmarks. DiffEval first generates a variety of prompts for each task within a benchmark representing multiple possible formulations per task. The framework then queries an array of LLMs using the generated prompts to obtain multiple outputs for every single task. As outputs will be code fragments, code-based metrics evaluate each fragment's functional and syntactic correctness to obtain a difficulty score per task. This difficulty score separates hard from easy tasks. Tasks can then be further analyzed using topic-modelling methods to group them into relevant topics. Finally, new tasks can be generated using the defined topics and leveraging identified hard tasks per topic.

The findings of this study are based on two coding task benchmarks and five LLMs for code. We show that while benchmark-wide metrics such as accuracy can give a general trend of a benchmark difficulty, they fail at assessing the difficulty of individual tasks contrary to the difficulty score proposed in DiffEval. In particular, the single prompt formulation for a task is limited when evaluating the tasks properly. Moreover, we observe that hard tasks are generally scattered around task topics, so there are no particular hard topics for LLMs, but rather hard tasks. Finally, using the identified topics, we generated 15 new tasks for both benchmarks for which the difficulty score is high as an example of DiffEval's capacity.

To structure the study in this chapter, we define four RQs:

RQd_1 *Does the level of information in a prompt correlate with the score of the generated code samples?* This RQ aims to validate the level of information transformation, rephrasing having already been shown to impact outcomes in the literature, by calculating the correlation between the level of information and the score obtained for the generated samples. If the level of information impacts the score obtained, we should obtain a positive monotonic correlation.

RQd_2 *Can DiffEval identify hard tasks in a benchmark?* In this RQ, we compare global evaluation metrics and the proposed approach when it comes to judging whether an individual task can be considered hard. This should translate to tasks labelled as hard by our approach, resulting in low accuracy when using global metrics. However, tasks labelled hard according to global evaluation would not necessarily translate to a high difficulty score using our approach.

RQd_3 *How difficult are tasks contained in the studied benchmarks for LLMs?* This RQ ex-

plores the different tasks within the analyzed benchmarks using our proposed difficulty score. We do so regarding the task distribution in terms of difficulty scores and how they are distributed across different topics. Said topics will be calculated using topic-modelling approaches to cluster tasks (hard and not hard) together to potentially identify topics that are inherently difficult for LLMs.

RQd_4 *Can DiffEval support the generation of targeted new difficult tasks?* In this RQ, we determine if generating further new hard tasks is possible by leveraging previously identified topics and hard tasks. In particular, the approach uses methods referenced in the literature to generate new tasks but accounts for the fact that we aim for specific hard tasks. If the approach is effective, the new tasks generated should be hard according to our approach, i.e., having a high difficulty score.

7.2 Motivating Example

Consider the following task to implement the function "cycpattern_check".

```
1 def cycpattern_check(a , b):
2     """ Write a function named 'cycpattern_check' that checks if the
    second word or any of its rotations is a substring of the first word.
    """
```

Listing 7.1 A programming task example.

To assess an LLM, one would generally run this prompt either using greedy decoding or temperature sampling [100, 236–238], and then test the outcome against a set of predefined tests to evaluate correctness. Multiple LLMs could even be used to offset the potential impact of the particular training/architecture on the evaluated correctness. Doing so with a greedy decoding using the 5 LLMs we use in our experiments for querying, results in all models except one failing on this task. In this context, should the task be considered hard? We show with *DiffEval* that this is not the case for this task. Indeed, using our framework, all models except one indicate a low difficulty score for the task with a low difficulty averaged over all LLMs. As it turns out, rephrasing this basic prompt slightly or providing small additional contextual information helped the LLMs to address the task easily. As such, while the above *prompt* could be considered hard, the *task* per se is not.

7.3 Methodology

In this section, we discuss the methodology followed to propose *DiffEval*. After an overview of the framework, we describe how we generate different prompts for programming tasks. We then delve into the details of leveraging LLMs to generate code for our prompts, measuring the difficulty of tasks for LLMs, identifying hard coding tasks, and generating new hard tasks.

7.3.1 General Overview of *DiffEval*

DiffEval uses existing benchmarks to operate. Such benchmarks generally comprise an ensemble of *tasks*, represented by a single *prompt* per task. In our study, we distinguish between *task* and *prompt*. The *task* is the essence of the functionality to be implemented. For instance, "Quicksort" or "Fibonacci sequence" are examples of such tasks. A *prompt* is a way to formulate the task so it can be understood and fed to an LLM. For instance, "Write a function to calculate the n-th element of the Fibonacci sequence." or "Create a function to return the n-th element of the Fibonacci sequence. The Fibonacci sequence is defined as etc." are two instances of *prompts* for the *task* "Fibonacci sequence". As such, each *task* can be expressed by many different (even infinite) *prompts*, including different wordings and amount of contextual information about the task. *DiffEval* aims to evaluate the difficulty of those *tasks* for code LLMs in order to generate targeted benchmarks to probe potential hard tasks of LLMs.

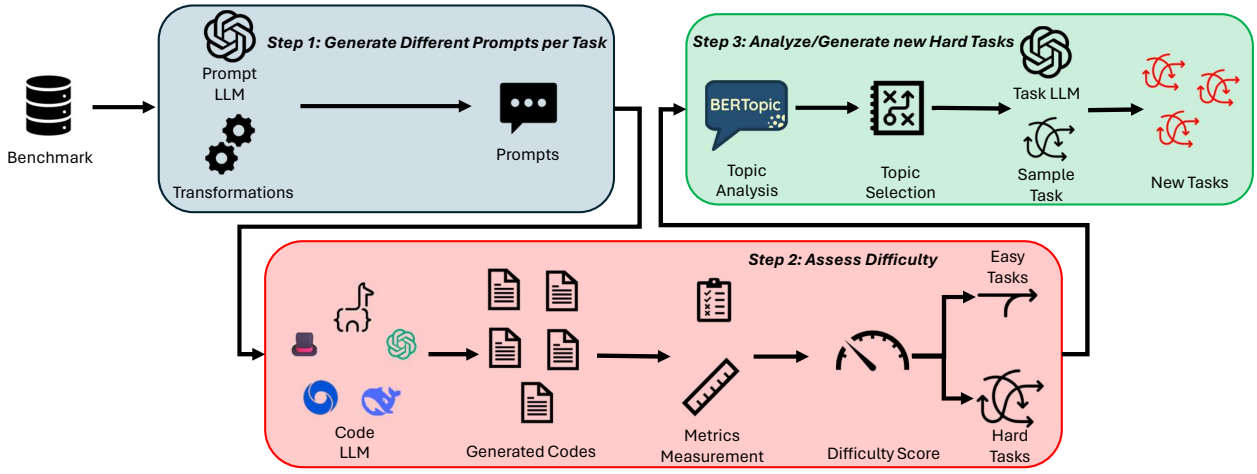


Figure 7.1 Overview of *DiffEval* Framework.

A general overview of *DiffEval* is given in Figure 7.1. *DiffEval* is operated in three steps: ① First, transformations on the prompts are identified. Those transformations are any dimension that can lead to various prompts for an LLM while keeping the essence of the

task. In a way, such transformations can be seen as metamorphic transformations [40] as they should not alter the semantics of the task. Our study focuses on two transformations: *Rephrasing* and *Context Information*. Those transformations are then used in conjunction with each task to have a *Prompt LLM* generate prompts following the transformations' specifications. ② Secondly, the obtained prompts are fed into several *Code LLM*s to generate code. Those LLMs aim to generate code for each given prompt and assess how LLMs fare when confronted with the different prompts. Multiple LLMs are used to encompass the diversity of architecture and training settings that could react differently to a particular prompt. Several metrics are extracted from the generated code that will help us obtain a *Difficulty Score* for each task in the benchmark. This score will further help us classify tasks into easy and hard ones. ③ Finally, "hard" tasks are analyzed to generate further targeted new tasks based on those topics. In particular, such tasks can be clusterized into tasks of similar semantics (e.g., "sequence" tasks, and "sorting" tasks), i.e., *topics*, and fed into a *Task LLM*. This allows us to generate new tasks similar in semantics to the identified hard tasks (e.g., new "sequence generation" based tasks), effectively generating a sub-benchmark of targeted tasks, which can bring a finer analysis of LLM shortcomings for code generation. In the following, we will describe each step in more detail.

7.3.2 Generating Different Prompts based on Transformations

The first step involves choosing transformations to generate different prompts for the task. We want those transformations to introduce variabilities in our prompts to represent the many ways an LLM could be prompted for the same task. However, those transformations should not introduce changes that alter the semantics of the tasks. In our case, we use two types of such transformations, but any number could be used and combined.

First, we use the *Context Information* transformation. Indeed, a straightforward way to act on the variability of a prompt is to disclose or withdraw certain information in the prompt while not changing the task. EvoEval [239], for instance, proposes also to act on the difficulty. However, they do so by adding/removing constraints in the prompt, thus modifying the semantics of the task, which is not desirable in our case. In our study, we instead propose to divide the information provided in the prompt into three levels. Those three levels of prompts are generated incrementally using the previous level as a starting point. We assume that prompts with more information should be more likely to lead to the correct code for LLMs. The levels are defined as:

- *First level*: contains a minimal amount of information that is the inputs/outputs of the task as well as a high-level description of what is intended. We could use the

original prompt in benchmarks for each task as the first level, yet Siddiq et al. [240] showed that there can be inconsistencies across tasks' prompts in benchmarks. Thus, we prefer to generate a new prompt with similar formatting across all tasks and levels to reduce biases. Moreover, this level is the closest to the original benchmark prompt and contains the same level of information regarding the tasks.

- *Second level*: further adds a description of the targeted function in natural language using available oracle code in the prompt. However, it does not include any direct reference to the function (variable names, helper functions used, etc.) unless it was explicitly mentioned in the original prompt of the benchmark.
- *Third level*: complete the prompt with direct references to the function, such as variable names and helper functions used similarly to a pseudo-algorithm but in natural language.

We present an example of different prompt levels obtained for a task in Listing 7.2.

```
1 def unique(l: list):
2     Level1: """ Write a function named 'unique' that returns a sorted list
      of unique elements from a given list. """
3     Level2: """ Write a function named 'unique' which takes a list as
    input and aims to return a sorted list containing only the unique
    elements from the input list. The function should first convert the
    list to a set to remove any duplicates and then convert it back to a
    list which should be sorted before returning. """
4     Level3: """ Write a function named 'unique' which takes an input list
    'l' and returns a list containing only the unique elements from 'l',
    sorted in ascending order. The function should first convert "l" into a
     set, using 'set(l)', to remove any duplicates, then convert this set
    back to a list, and finally return this list after sorting it in
    ascending order using 'sorted()'. """
5     return sorted(set(l))
```

Listing 7.2 Example of different prompt levels for the task *unique* from HumanEval+, with the oracle code at the bottom.

The second transformation used is *Rephrasing*, paraphrasing a given prompt. Indeed, how the prompts or instructions are formulated has been shown to impact LLM responses [162, 163, 241]. Thus, similarly to these studies, we try to see if different ways of rephrasing a given prompt for an LLM can affect how the LLM addresses the task. We will use a similar approach to rephrase our prompts as Gonen et al. [242], prompting an LLM to rephrase a

given text. Other techniques could assist in rephrasing, such as Chains-Of-Thoughts [243], but assessing the effectiveness of different rephrasing techniques is out-of-scope of this study.

Using those two transformations, we come up with general templates that can be used to generate prompts (see replication package [244]). Such general templates are tailored for each task in the benchmark under consideration. Those templates are fed to a *Prompt LLM* (see Figure 7.1) to generate different prompts for a given task. The way prompts are generated is as follows: we start from the original docstring to have a common starting point for all prompts. Then, we apply the template to obtain different prompt levels with *Context Information* for a given task. Then, we apply the obtained prompts to the *Rephrasing* template. In our case, we first remove the examples from the original prompt, if any, before using our templates with the *Prompt LLM*. While removing examples can decrease the performance of LLM [245], this allows for better control over the level of information we inject into our prompts. Indeed, examples can vary from simple basic examples to showcasing corner cases of the task. As such, this might artificially bias the output of LLM and the potential difficulty of the task because of the quality of the examples.

At each step of our generation, that is after using each template on the prompts, we do a sanity check by manually checking them. This is accomplished by the first author and independently cross-checked by the second author. We make sure that the *Context Information* levels are respected and the *Rephrasing* does not alter the semantics of the tasks. This is done to avoid tasks that are artificially considered hard further down the line because of misleading prompts for the code LLM and not because of the task itself.

7.3.3 Assessing difficulty of tasks

The second step is to evaluate the difficulty of each task. To do so, we leverage several *Code LLM* that will serve as our evaluators for code generation. For each prompt per task, each LLM is asked to generate code with varying seeds using the sampling mode of LLMs. The rationale is that most LLMs, especially non-open-source ones, are used with this non-greedy approach to generate code. Moreover, using the sampling method for the generation allows us to have a wider variety of generated code to evaluate the difficulty of each task. Indeed, we assume an easier task should not be as impacted by the stochasticity of the generation compared to a harder task. The obtained code is then processed and executed. Sample codes were then executed against available test inputs in the benchmark.

Once we have obtained all code for all prompts of all tasks and executed them against tests, we collect different metrics. We are interested in two metrics: the *Correctness* and the *Similarity to Correct Code* of a code sample. *Correctness* quantifies if a given code sample

is correct or not using available test cases. As such, it assesses the functional correctness of a code sample. Formally, the Correctness Cor for a code sample s is defined as:

$$Cor_s = \begin{cases} 1 & \text{if } \forall (i,o) \in T, s(i) = o \\ 0 & \text{else} \end{cases} \tag{7.1}$$

where T is the set of test cases symbolized by an input i and an output o. That is, the code sample passes all test cases, i.e., gets a value of 1. Intuitively, we expect a harder task to contain fewer prompts leading to correct codes across rephrasing and different levels of context information.

Nonetheless, code samples on a task can be wrong to different degrees depending on the bugs affecting the code sample [246]. Indeed, a code sample could be wrong because of a missing corner case (e.g., forgetting an initial condition), or it could follow a false logic and not implement the desired tasks. One way to assess this aspect would be to quantify the *proportion* of test cases for which a code sample s is correct [247]. However, depending on the test set, this can lead to a biased measure. For instance, if a test set has more test cases assessing one corner case rather than another one, some code samples might end up with a higher proportion of passed tests artificially. Instead, we propose to use classical similarity metrics to measure the distance between a code sample and the *the most similar* correct code sample. To do so, we will rely on the *CodeBLEU* [39] metric to measure the similarity, which measures the syntactical correctness of the code sample. We chose CodeBLEU as it is a widely used metric in code generation tasks [248–251]. Thus, the Similarity to Correct Code Sim for a code sample s is defined as:

$$Sim_s = max_{sc}\texttt{CodeBLEU}(s, sc), C_{sc} = 1 \tag{7.2}$$

where sc are correct code samples according to *Correctness*. Note that, in practice, we have at least one correct code as a reference, which is the oracle code of the task in its benchmark. If s is correct, the similarity is 1 (closest is itself). This allows us to assess, when a code sample is incorrect, *how* similar it is to a correct code. The *Correctness* and *Similarity to Correct Code* are then used to evaluate a code sample separately or in a *Composite* manner by aggregating the scores.

Finally, we use those metrics to compute a score per code sample, which in turn helps to assess the difficulty score of the *task* itself for a given LLM. We used different transformations in this study: *Rephrasing* and *Context Information*. So, for instance, we expect an LLM to be more likely to generate a correct code if given a prompt with more context information

(of a higher level) than a prompt of a lower level. To represent this, the scores of the code samples obtained for each prompt can be weighted depending on the transformation. In our case, as we apply the *Context Information* before the *Rephrasing*, the difficulty score R_i of the i^{th} *Rephrasing* is defined as:

$$R_i = \sum_j s_j \times score_{s_j} \tag{7.3}$$

where $score_{s_j}$ is the score obtained for the j^{th} code sample using the i^{th} rephrasing, as we generate multiple code samples for a given prompt through sampling generation. s_j are weights that give more or less importance to certain code samples. In our case, all samples are equally important, so all s_j equals 1 over the number of samples. Then, we calculate the score L_i for a given level of prompts i:

$$L_i = \sum_j r_j \times R_j \tag{7.4}$$

where r_j are weights used to give more or less importance to a certain rephrasing. In our case, all rephrasings are equally important, so all r_j equals 1 over the number of rephrasings. Finally, the difficulty score of the i^{th} task in the benchmark for a given LLM is calculated as:

$$Diff_i = 1 - (\alpha \times L_1 + \beta \times L_2 + \gamma \times L_3) \tag{7.5}$$

where α, β and γ are weights over the scores obtained for each level. Regarding the choice of parameters, setting equal weighting for the rephrasing would not necessarily be a sound choice. Indeed, as the prompts of lower levels have less information than those of higher levels, they are more likely to lead to a lower score, which could artificially decrease the difficulty score. Thus, to counteract this effect, we tip the weights so as to make the scores of the prompts of the lower levels more relevant. Indeed, an easy task should have prompts of a lower level, leading to more correct codes than a hard task. In our case, we empirically set the weights as follows: $\alpha = 0.5, \beta = 0.3, \gamma = 0.2$.

Finally, once all the $Diff$ scores have been computed for all tasks and LLMs, the score of a given task is simply the average over all LLMs. One can then use the score to determine which tasks are hard and which are not. To do so, we will use a threshold of 0.5. The choice of the threshold is motivated by our definition of the score for each code sample: a score of 0.5 would mean, in the most extreme case and for the *Composite* score, that all generated code samples for a task are incorrect but are very similar to a correct one. In practice, a score higher than 0.5 for a task would reflect a higher number of incorrect codes with a lower

similarity, pointing towards a hard task.

7.3.4 Analyzing and Generating new Hard Tasks

Once the hard tasks have been identified, the third step is to generate new tasks based on the identified hard tasks. To do so, we propose to leverage a topic modelling approach with BERTopic [252], as was used in similar studies when dealing with natural language data [253–255]. For a given benchmark, the topic model clusters all tasks (easy and hard) using the prompts of Level 1 before rephrasing. We use this specific prompt type to have similar formatting across tasks while being as close as possible to the original prompt, as mentioned in Section 7.3.2. As the process will inherently label some tasks as "noise", noisy tasks are not considered in the analysis. Then, for each topic, the proportion of hard tasks among all tasks in the topics is calculated, and the topics with the highest proportion of hard tasks are analyzed. Those topics constitute tasks with a similar functionality/goal, such as tasks dealing with sequence generation. Finally, we aim to generate more hard tasks leveraging the tasks from those selected *topics* with a *Task LLM* to generate additional *new* tasks. To do so, we leverage the Self-Instruct approach [256], which allows the generation of new tasks based on a subset of sampled tasks. Nonetheless, according to our observation, just providing easy and hard tasks to the *Task LLM* and asking it to generate more hard tasks did not lead to meaningful results. Indeed, the small number of tasks per topic makes it quite impossible to determine what constitutes a hard task. Instead, we use a combination approach, where we ask the *Task LLM* to augment a hard task by adding a constraint or by adding a constraint from a hard task to an easy one. This ultimately reduces the diversity of tasks generated but increases the likelihood of the generated tasks being hard. The newly generated tasks constitute a smaller sub-benchmark of targeted tasks related to the *topic* that can be used further to assess LLMs corner cases for code generation tasks.

7.4 Experiments

7.4.1 Hyperparameters used

To have different prompts, we generate 3 levels and 6 rephrasings per level. We chose those parameters as the trade-off between generating diverse enough prompts while limiting the amount to generate. For a benchmark of n tasks, this effectively means we have to generate $18 \times n$ prompts. We further ask each *Code LLM* to generate 5 code samples per individual prompt, resulting in $90 \times n$ code samples for each *Code LLM* to generate. As an example, for a benchmark containing 200 tasks, this process yields **18,000** generated code for our

evaluation ***per*** *Code LLM*. We make available all the prompts and code generated in our replication package [244].

7.4.2 Benchmark used

In our experiment, we chose two benchmarks: HumanEval+ [257] and ClassEval [258]. Those benchmarks were chosen because they cover different tasks and dependency levels. HumanEval+ is widely used as a standard benchmark in code generation [259]. It is an extended variant of HumanEval [100], that contains more tests to assess the correctness of code samples. The benchmark consists of 164 handwritten Python programming problems with a function signature, docstring, body and several unit tests. HumanEval+ tasks are at the function level dependency as they do not require external libraries besides Python standard library or class/file contexts to be operated. For the original prompt, we use the docstring as given in HumanEval+.

ClassEval is a benchmark of 100 handcrafted classes coded in Python. Each class contains a description and on average 4 methods each with a docstring. We consider a task to be one of the 400 methods of ClassEval. Contrary to HumanEval+, ClassEval tasks are more similar to practical code used in projects and rely on a class context to operate. ClassEval proposes three prompting strategies: holistic, incremental and compositional. In our case, we use the compositional strategy, providing the class context to the LLM and asking it to complete one method of the class, only modifying the method's docstring to complete through our crafted prompts. We do not use the other generation approaches as it reduces our control over the prompts and could introduce biases: the holistic approach, as it implies asking the LLM to generate the whole class from scratch, can impact the performance artificially. Similarly, the incremental approach requires reusing codes generated by the LLM for other methods when generating new ones, which could also alter the difficulty. As we need to generate many code samples and prompts, we reduce the number of tasks to evaluate by stratified sampling at the 95% confidence interval. For 400 methods, this gives us 197 methods to sample, which we rounded up to 200. Those 200 methods are sampled equally across classes, resulting in 2 methods per class available. Methods chosen are available in our replication package [244].

7.4.3 Models used

In our experiments, we use GPT-4 [260] as both the *Prompt LLM* (to generate the different prompts) and the *Task LLM* (to generate new hard tasks). We do so as GPT-4 is the state-of-the-art LLM, and so should provide answers with the best quality, which would require minimum human supervision. For the *Code LLM*, we use 5 LLMs with different architecture:

CodeLLama 7B [261], MagiCoder 6.7B [262], DeepSeekCode 7B [263], CodeGemma 7B [264] and GPT-3.5 [5]. We have not used GPT-4 as *Code LLM* to avoid biases as it generates the prompts fed to the *Code LLM*. For all models, we used their instructions-tuned versions so the models could handle the different prompts properly. This gives us a diverse array of architecture and training procedures. For all code generation in *DiffEval*, as we are using the sampling generation, we fix the temperature hyperparameter to 0.8 as it is one of the temperature settings commonly used in other similar studies [100, 257, 261, 265]. In all cases except GPT models (for which we use OpenAI API), we use the HuggingFace [266] implementation of the models to allow for replication. We compute Pass@1 [100] (see Chapter 3.3.1) for verification with the samples obtained across levels for a given task. Results are reported in Table 7.1. The ranking of the LLMs' performance follows the assessment obtained on the original benchmarks.

Table 7.1 Per levels Pass@1 with Temperature = 0.8.

Code LLM	HumanEval+			ClassEval		
	Level 1	Level 2	Level 3	Level 1	Level 2	Level 3
CodeLLama	35.69%	47.03%	65.44%	46.30%	58.98%	73.07%
DeepSeek	59.98%	69.29%	81.30%	56.82%	67.2%	77.90%
MagiCoder	56.38%	67.99%	81.91%	56.68%	65.57%	76.60%
CodeGemma	42.90%	57.09%	72.46%	49.41%	60.77%	73.68%
GPT3.5	61.97%	72.76%	84.86%	60.73%	72.73%	82.33%

7.5 Results

7.5.1 RQd_1: Does the level of information in a prompt correlate with the correctness score of the generated code samples?

The goal of this RQ is to assess whether the level of information in the prompts impacts the generation of the code by the LLM in such a way that is quantifiable by our metrics. As reported in Table 7.1, the Pass@1 rate seems to increase with the *Context Information*. We now set out to show there is a correlation between the scores computed by our metric and this *Context Information*. To do so, we use the scores of the generated code for each rephrasing (see Equation 7.3) and compare it to the prompt level it is from. To see the impact of both the *Correctness* and *Similarity* metric, we will compute each metric individually and the composite of the two. As we introduce gradation in the injected information in the prompts through the *Context information*, we expect to observe a correlation between the level of information and the score obtained (i.e., the more information in the prompt, the higher

the score). We use the Spearman-ρ [267] to calculate the correlation, as we are interested in a monotonic increase between the two variables. We give the results for each *Code LLM* independently and for the averaged over all LLMs. Results are reported in Table 7.2.

In Table 7.2, we obtain a significant positive correlation for all cases, indicating that increasing the amount of information in the prompt generally improves the LLM-generated code in terms of the score for all metrics. The Composite score (i.e., *Correctness* + *Similarity*) yields the best correlation overall on HumanEval+ while plain *Similarity* is slightly better on ClassEval. As we have a positive correlation, higher scores generally mean higher for a level 3 prompt than a level 1 prompt, so they account for our *Context Information*. As the Composite metric seems to be better on HumanEval+ and is tied up on ClassEval, we consider only this metric in the following to calculate our difficulty score.

We note that the correlations are around 0.2 to 0.4, signifying a positive significant but weak to moderate relationship. We suppose the reason comes from the variability across tasks. Indeed, due to the particular difficulty of each task or rephrasing, there might be some variability within the tasks that could impact our outcome. For instance, easier tasks are more likely to have a higher score for Level 1 prompts, while more difficult tasks can end up with a low score for Level 3 prompts. This can impact the outcome of the Spearman-ρ when considering all tasks at once. To account for this, we calculate the Spearman-ρ for each task individually before averaging the obtained correlations. As averaging correlations directly might lead to biases, we follow the recommended approach [268, 269] of converting the Spearman-ρ correlations using the Fisher Z-transformation, averaging the results and converting it back to a Spearman-ρ. Doing so, we obtain Spearman-$\rho > 0.5$ in all cases, even reaching 0.70 and 0.64, respectively, for HumanEval+ and ClassEval when using the scores averaged overall LLMs. This results in moderate to strong relationships. Our *Context Information* helps LLMs generate correct code when we account for task variability, and so is valid to calculate the difficulty score.

Findings 1: The Composite metric highlights the difference between prompts with different levels of information. In particular, this score exhibits significant positive correlations with the level of information in the prompt for all LLMs as well as the average of all LLMs.

7.5.2 RQd_2: Can DiffEval identify hard tasks in a benchmark?

This RQ aims to show that *DiffEval* can reliably identify hard tasks in a benchmark. To do so, using greedy decoding (i.e. Temperature = 0.0), as done in the-state-of-the-art leaderboard

Table 7.2 Correlation (Spearman-ρ) between the scores and the *Context Information*. All correlations are significant (p-value < 0.001). HE: HumanEval+ and CE: ClassEval.

Code LLM	Composite		*Correctness* Only		*Similarity* Only	
	HE	CE	HE	CE	HE	CE
CodeLLama	0.36	0.27	0.27	0.26	0.32	0.28
DeepSeek	0.30	0.22	0.23	0.22	0.26	0.23
MagiCoder	0.37	0.21	0.28	0.20	0.31	0.21
CodeGemma	0.33	0.28	0.30	0.27	0.36	0.29
GPT3.5	0.25	0.24	0.23	0.24	0.27	0.25
Overall LLMs	0.40	0.27	0.37	0.26	0.36	0.28

[236, 237] for each of 5 LLMs of our study, we evaluate each task using a single prompt to see if each LLM outputs a correct code using available tests. For the single prompt used, we chose the Level 1 prompt obtained before rephrasing for each task, which is the closest to the original prompt of each benchmark. We did so for a similar reason as mentioned in Section 7.3.2, i.e. removing potential formatting issues and providing a fair comparison across tasks. For greedy decoding, we consider a task hard for an LLM if it does not generate a correct output. We consider a task hard for all LLMs if no LLM manages to generate a correct code for that task. We use similar settings for easy tasks. At the same time, we collect tasks that are labelled as hard/easy by *DiffEval* using our difficulty score (see Equation 7.5), which is when the score is above/below 0.5. Using those sets of tasks, we can compute the overlap between the sets, e.g., the tasks that are hard according to *DiffEval* but not for the greedy decoding definition, for a given LLM or all LLMs. This allows us to see if the sets of tasks are similar across approaches. Then, we compute the accuracy of the greedy decoding over the tasks by splitting them into two sets of tasks, using the sets of hard/easy tasks of *DiffEval*. Similarly, we compute the median of the difficulty score of *DiffEval* obtained on the tasks by splitting them into two sets, based on the set of hard/easy tasks from the greedy assessment, for each LLM. Intuitively, we should see a drop in performance of the accuracy of the greedy decoding on the tasks that are noted hard by *DiffEval* and an increase in performance for the tasks that are labelled as easy by *DiffEval* compared to the greedy accuracy of each LLM.

Computing the overlap between the sets of hard tasks for both approaches results in an overlap ratio ranging from 0.18 to 0.54 for HumanEval+ and from 0.49 to 0.55 for ClassEval, depending on the LLM. Similarly, the overlap for the sets of easy tasks ranges from 0.49 to 0.75 for HumanEval+ and from 0.66 to 0.75 for ClassEval, depending on the LLM. Considering all LLMs, the overlap on the hard tasks is 0.40 for HumanEval+ and 0.55 for ClassEval, while for the easy tasks, it is 0.33 and 0.46, respectively. Thus, the assessment of hard tasks

between the two approaches is dissimilar, while there are more similarities in the assessment of the easy tasks, at least per LLM. We then provide the results regarding computing the metric of each approach on the hard/easy tasks given by the other one. Results are reported in Table 7.3. For each row of the table, we get, for an LLM, the results (average difficulty score or accuracy) on both HumanEval+ and ClassEval benchmarks using the subsets of hard/easy tasks per approach. For instance, for CodeLLama on HumanEval+, the median difficulty score on the easy tasks of CodeLLama for the greedy approach "Greedy Easy", when the output for a task is correct, is 0.25. Similarly, for GPT3.5, the accuracy of the greedy approach on hard tasks according to *DiffEval* is 6.90%. Moreover, we also report the Accuracy of the greedy decoding over the whole benchmark for each LLM and averaged across LLMs.

Table 7.3 Median of the difficulty scores and Accuracy of the greedy decoding on the subsets of hard/easy tasks for each approach. For the difficulty score, we compute the median using *DiffEval* difficulty score on the easy/hard tasks of the greedy approach (*Greedy Easy*/*Greedy Hard*). For accuracy, we calculate it using the easy/hard tasks labelled by *DiffEval*. We also provide the accuracy of the greedy decoding on the whole benchmark (*Greedy Acc*).

Benchmarks	HumanEval+					ClassEval				
Code LLM	Greedy Easy	Greedy Hard	Greedy Acc	DiffEval Easy	DiffEval Hard	Greedy Easy	Greedy Hard	Greedy Acc	DiffEval Easy	DiffEval Hard
CodeLLama	0.25	0.55	37.20%	54.95%	15.07%	0.03	0.54	49.00%	66.90%	5.17%
DeepSeek	0.11	0.33	63.41%	67.12%	33.33%	0.01	0.52	58.00%	73.20%	8.51%
MagiCoder	0.16	0.35	65.85%	70.92%	34.78%	0.02	0.53	59.05%	75.64%	2.27%
CodeGemma	0.09	0.51	49.39%	65.79%	12.00%	0.05	0.53	54.00%	70.95%	5.77%
GPT3.5	0.03	0.48	64.02%	76.30%	6.90%	0.01	0.5	61.50%	75.63%	5.00%
Overall LLMs	0.11	0.51	55.97%	65.15%	11.43%	0.01	0.59	56.31%	71.37%	7.66%

In Table 7.3, when using *DiffEval* sets of easy/hard tasks, we see an increase/decrease in performances for both benchmark and all LLMs compared to the performance calculated over the whole benchmark (see Table 7.3 "Greedy Acc"). This shows that the tasks identified as easy/hard by *DiffEval* are indeed easy/hard, as the subset of hard tasks will result in lower accuracy and the subset of easy tasks in higher accuracy compared to the whole benchmark baseline. Secondly, we see that, for all LLMs, easy tasks for the greedy decoding (i.e., whether individual LLMs manage the tasks and, Overall LLMs, all LLMs manage the task) obtain a median of the difficulty score < 0.5, showing that most of those tasks are indeed easy according to *DiffEval*. Third, we get a median difficulty score fairly close to 0.5 for the set of hard tasks for the greedy decoding, the only exception being for DeepSeek and MagiCoder on HumanEval+ (0.33, 0.35). Thus, most tasks that look hard when using greedy decoding, i.e. pass/fail on a single prompt, are not hard according to *DiffEval*. This, however, was to be expected, as we saw in Listing 7.1, using a single prompt, a task can easily be hard

for one or more LLMs because of the formulation and/or underspecifications in the prompt. However, rephrasing the prompts or adding information can lead to easy tasks, something a single prompt can not tell us.

Findings 2: Using the correctness of the code over multiple LLMs on a single prompt is not reliable for identifying hard tasks. On the contrary, the difficulty score of *DiffEval* can identify hard/easy tasks in a benchmark.

7.5.3 RQd_3: How difficult are tasks contained in the studied benchmarks for LLMs?

Our goal in this RQ is to study the distribution of difficulty scores across tasks and the type of tasks that are difficult in the benchmarks under study. To do so, first, we use *DiffEval* difficulty scores for each task calculated in RQd_2, both for the *Code LLM* individually as well as all LLMs. By doing so, we can output the cumulative distribution of the difficulty score across tasks for analysis. Then, we use the BERTopic model described in Section 7.3.4. The topic analysis returns 17 topics for HumanEval+ and 21 topics for ClassEval. The first two authors manually check the tasks in those topics to provide a general name for the topic that is as representative as possible of the tasks within the topic. We then calculate the proportion of hard tasks in each topic.

The results for the cumulative distributions are given in Figure 7.2 for both benchmarks. On both the benchmarks under study, 67% of the tasks for HumanEval+ and 69% of the tasks for ClassEval, Overall LLMs, obtain a difficulty score of less than 0.4 and only 17% of the tasks for HumanEval+ and 24% of the tasks for ClassEval have a difficulty score higher than 0.5 thus making them hard tasks with our definition. In particular, 43% of all tasks for HumanEval+ and 64% of all tasks for ClassEval are easy for all LLMs. Thus, both benchmarks seem to have a similar difficulty profile, which echoes the accuracy presented in Table 7.1. The main difference comes from 1) ClassEval tends to have a high proportion of trivial tasks (< 0.1), around 40%, and most LLMs agree on that while on HumanEval+ it is much more nuanced, 2) CodeLLama and CodeGemma end up having better performance on ClassEval. We suppose the difference comes from the fact that HumanEval+ has more algorithmic tasks. In contrast, ClassEval tasks are more related to general development tasks, which could benefit those two LLMs, and 3) ClassEval has more difficult tasks overall. This echoes the subjective assessment that ClassEval is "harder" than HumanEval+, as it relies on a class context. 4) We highlight that, while the accuracy over both benchmarks seems similar (see Table 7.1 and Table 7.3 "Greedy Acc" for the same Code LLMs), the

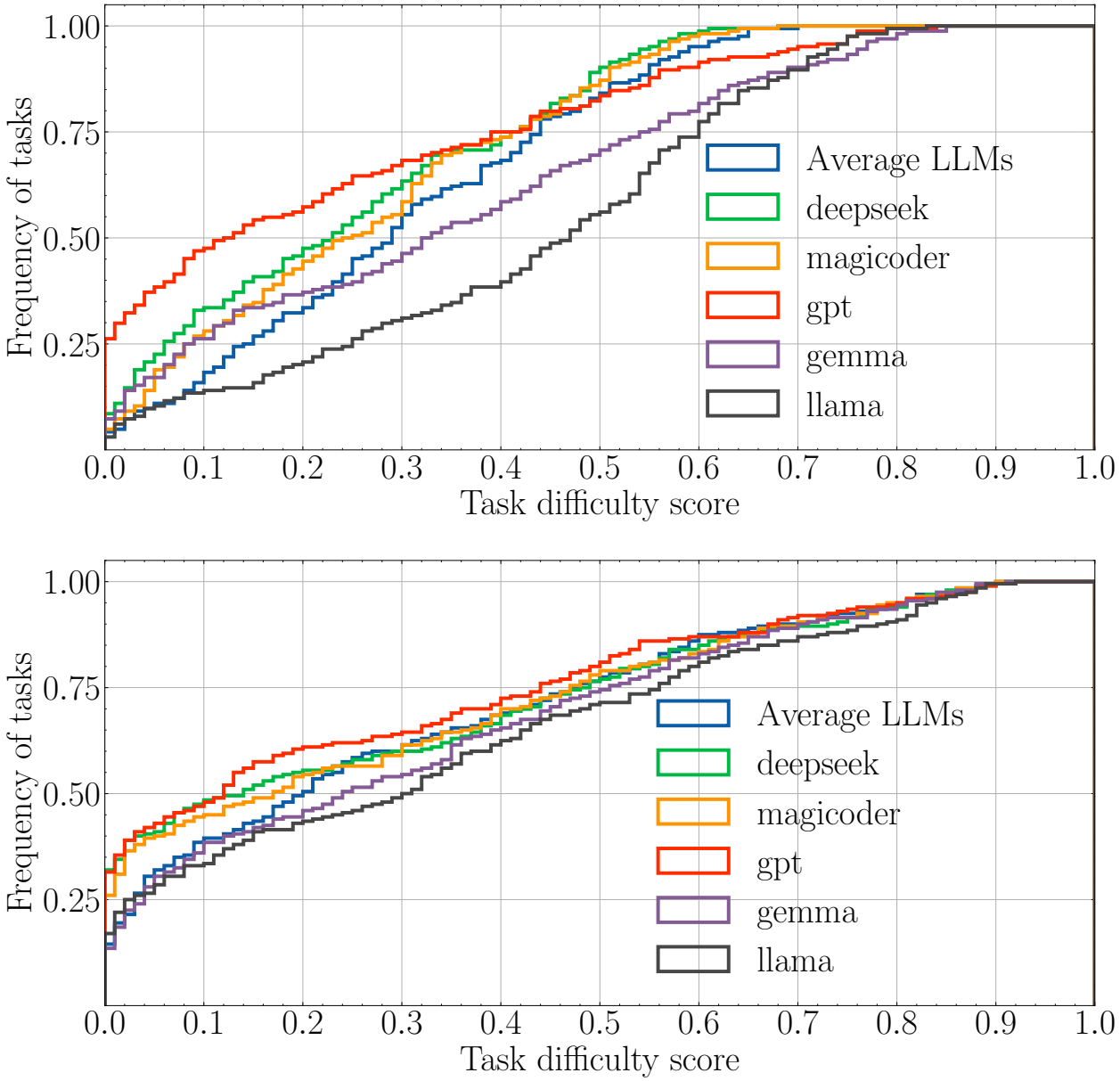


Figure 7.2 Proportion of tasks in HumanEval+ (top) and ClassEval (bottom) with a given difficulty score for each individual LLM and for the average.

distribution of difficulty of tasks between two benchmarks are different: computing a two-sample Kolmogorov-Smirnov test [270] to assess if the distributions overall LLMs are the same, leads to a p-value < 0.01 and so we can indeed conclude that both distributions are different.

We then extract topics using BerTopics as described in Section 7.3.4 to group tasks into topics and analyze them. The output is given in Figure 7.3.

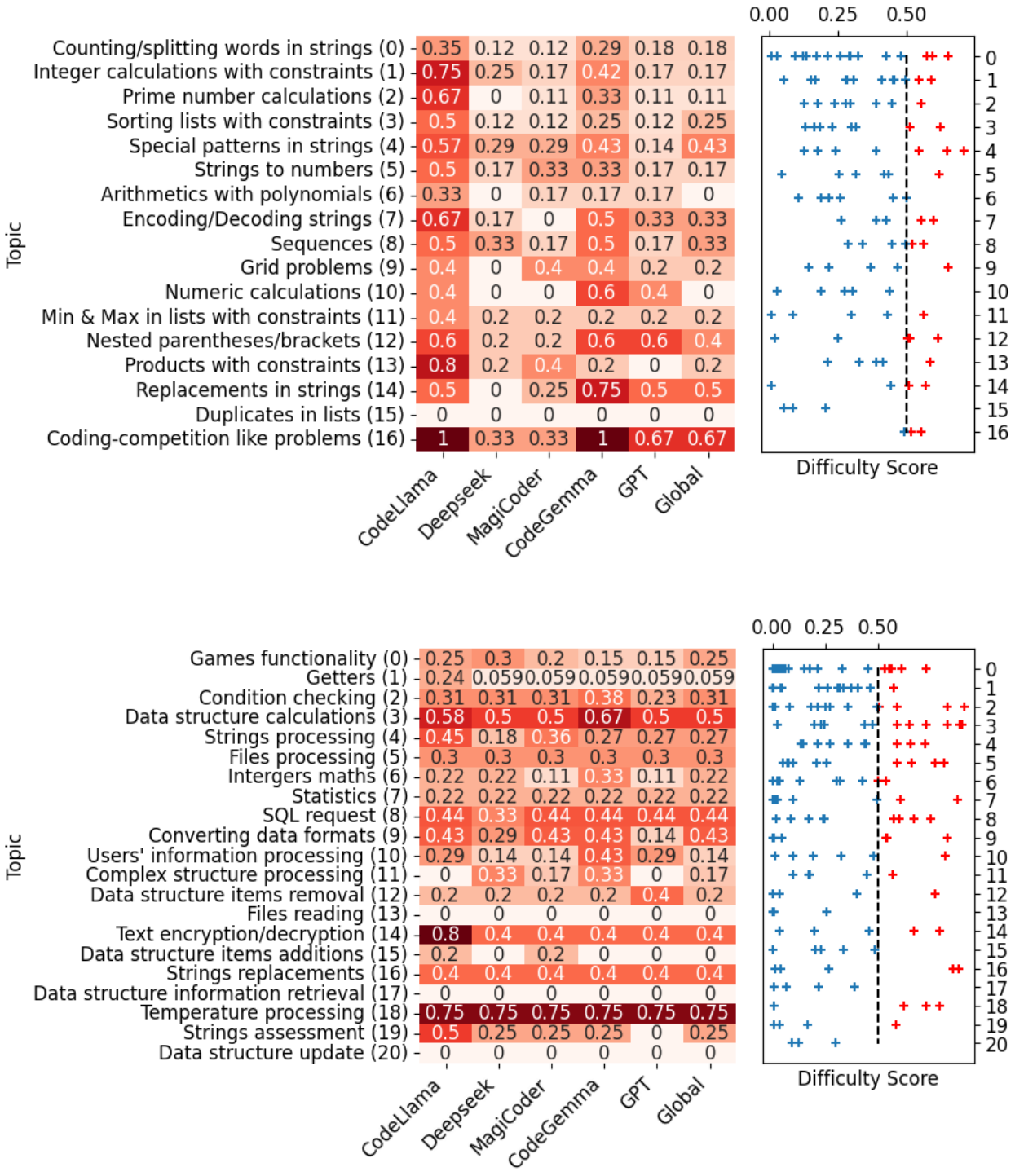


Figure 7.3 Topics of HumanEval+ (top) and ClassEval (bottom). For each, we give (left) the proportion of hard tasks in a topic and (right) the average difficulty score for each task inside the topics across all LLMs. Red crosses are hard tasks.

One can make several observations from the obtained results. First, not all LLMs encounter the same difficulty depending on the tasks. For instance, GPT has more trouble on Topic 12 in HumanEval+ but less so on Topic 11 of ClassEval compared to Deepseek, as highlighted by the higher proportion of hard tasks in this topic and the average score over the tasks. This indicates that each LLM has strengths and weaknesses originating from its architecture and the data on which it was trained. While the prompts we use could affect this outcome, the fact that other LLMs have, on the same topic, opposite behaviour instead suggests that the root cause is mainly due to training and/or models' architecture. Secondly, some topics that are easy/hard across LLMs highlight some general behaviours. For example, Topic 15 of HumanEval+ or Topic 13 of ClassEval has no hard tasks in all models, on top of the individual tasks having difficulty scores lower than 0.25. On the other hand, Topic 16 of HumanEval+ and Topic 3 of ClassEval have a higher proportion of hard tasks for all LLMs. ClassEval seems to have more topics with a higher proportion of hard tasks compared to HumanEval+ but, at the same time, a higher proportion of topics with no hard tasks. Nonetheless, based on the extracted topics, it does not seem to be topics where all hard tasks are concentrated. On the contrary, for both benchmarks, the topics are scattered across the different topics. This further motivates us to use a combination of hard and easy tasks from a similar topic to generate new tasks: 1) as topics are rarely only hard or easy, combining both easy and hard tasks or augmenting hard tasks, for which we already have a result, increase the likelihood of re-obtaining a hard task, 2) by leveraging topics, we can easily make use of the thematic similarity between tasks to combine/augment them. This, for instance, prevents potential arbitrary transformation when combining tasks that are not directly related, which can have an unpredicted impact on the LLMs.

Findings 3: At a high-level, *DiffEval* difficulty score profiles of both benchmarks are similar, which echoes the similar Accuracy obtained in the Greedy decoding. However, ClassEval exhibits more hard tasks than Humaneval+ (24% against 17%). While some topics exhibit a high proportion of hard tasks inside the benchmarks, especially on ClassEval, most hard tasks are generally scattered across different topics.

7.5.4 RQd_4: Can DiffEval support the generation of targeted new difficult tasks?

```
def solve_with_vowel(s):
    def vowel_change(ch):
        return ch if ch not in "aeiouAEIOU" else chr(ord(ch) + 2)
    ans, has_letter = "", False
    for ch in s:
```

```
        if ch.isalpha():
            has_letter = True
            ans += vowel_change(ch.swapcase())
        else:
            ans += ch
    return ans if has_letter else s[::-1]
```

Listing 7.3 Oracle code of a new task generated. In yellow, we highlight the change compared to the original hard task used for generation. Changes are made to be minimal so the task is as similar as possible to an already established hard task.

Finally, this RQ aims to show that, using identified hard tasks, we can further generate new hard tasks targeted at specific problems. We do not seek to be systematic but merely to show that, using identified hard tasks and topics, it is possible to generate new tasks. To do so, we use the result from the RQd_3 (i.e., the topics) and select the first three topics in terms of the number of tasks, with at least a third of the tasks being hard. We do so to have sufficient LLM-easy and hard tasks for the *Task LLM* to provide meaningful new tasks. For each topic, using the *Task-LLM*, we generated tasks, checking each of them manually for validity, until we got 5 different new tasks using the process described at the end of Section 7.3.4. Then, we manually write an oracle code and unit tests for this task. When possible, we try to remain as close as possible to the oracle code of the base task used in the generation process. Finally, we assess the newly generated tasks to the same Step 1 and Step 2 of *DiffEval* to measure the difficulty score. We thus obtain new tasks for each of the picked topics that are assessed. We present an example of Oracle code from a newly generated task using the *Task LLM* in Listing 7.3.

Table 7.4 Average score for each of the 5 newly generated tasks for each topic. We highlight in bold the tasks that end up being easy.

HumanEval+	Task 1	Task 2	Task 3	Task 4	Task 5
Topic 3 - Patterns	**0.47**	0.65	0.59	0.73	0.53
Topic 7 - Encoding	**0.49**	0.68	0.65	0.86	0.72
Topic 8 - Sequence	0.76	0.78	0.76	0.73	0.73

ClassEval	Task 1	Task 2	Task 3	Task 4	Task 5
Topic 3 - Structure Calculations	0.78	0.81	0.88	0.88	0.71
Topic 8 - SQL requests	0.65	0.79	0.65	0.68	0.67
Topic 9 - Data format	0.85	**0.28**	0.87	0.77	0.84

The results for the difficulty score obtained for the newly generated tasks are given in Table 7.4. Most tasks generated following our approach also turn out to be hard. We note that a

few of them are below the 0.5. We see two possibilities for this: 1) either the base task was easy and we added a constraint from a hard task, which is not what makes the task hard in the first place, or not completely, 2) adding a small constraint to a hard task will modify the prompt and so, we might end up with some sets of prompts that are more favourable for the generation of codes for LLMs. However, the difficulty score is fairly close to 0.5, illustrating that the task is far from trivial. The only task with a fairly low score is the task present in Topic 9 for ClassEval. This task was obtained by adding a hard constraint from an easy task. Yet, as the constraint was not applicable perfectly, we added to relax it in the implementation, which might explain the low score. In particular, the new task-generation process can vary across topics and benchmarks. For instance, while generating a sequence with added constraints on HumanEval+ is simple for the *Task LLM*, coming up with new methods for already established classes on *ClassEval* can be quite hard, especially when the methods to use as a base are very different from each other. In practice, this results in the large majority of the tasks slightly modifying a precise hard task. However, while not perfect, simply reusing identified hard tasks can be a simple way to generate new tasks focused on specific topics.

Findings 4: Using identified topics and hard tasks rated by *DiffEval*, we can generate new tasks. Those new tasks are similarly hard, showing it is possible to generate new hard tasks using our approach.

7.6 Discussion

With *DiffEval*, we show that assessing the difficulty of a task from the point of view of LLMs is feasible, as well as identifying hard tasks. In particular, hard tasks identified with *DiffEval* are truly hard as they are more resilient to the potential variability of the prompt, contrary to assessing using a single prompt. While relying on human judgment to define a task difficulty could be used, it is unlikely to work for LLMs as they process code and task instructions differently than humans [271]. Ouyang et al. [247] showed that the difficulty assessed by humans on code competition only weakly correlates with the test pass rate variance across code samples generated by ChatGPT. Thus, it is likely that human assessment or related metrics might not be accurate for estimating LLMs' task difficulty contrary to the difficulty score of *DiffEval*. Our proposed approach, while applied in experiments to code generation, can be extended to other code-related tasks. Indeed, the transformations we used are easily transferable to other paradigms, and new transformations can be defined.

Analyzing the tasks labelled as hard, we observe that LLMs tend to disregard certain parts

of the instructions, which can lead to certain test cases failing, thus making the code incorrect. That is why similarity helps (i.e., code is incorrect but very similar to a correct one) (Finding 1). This phenomenon happens generally as the number of constraints in the prompt (and in the function) increases or if the prompt becomes longer. This explains why coding-competition or algorithmic-like problems in both benchmarks, which require many descriptions to express the many operations and constraints, generally lead to a higher difficulty, as the LLM is overwhelmed with information. This echoes Ouyang et al. [247], as they showed that description length impacts the test pass rate in ChatGPT. This, however, can not on its own explain why difficulty could be higher as Level 3 prompts tend to have a better score and are longer than lower-level prompts. We also note that corner cases must be explicitly mentioned; otherwise, LLM might miss it. For instance, stating when checking a date that February can have 29 days or when checking a condition in a string that the string can be empty, can improve the generation. This type of information is generally omitted or implicit in benchmark prompts (and so in our Level 1 prompts), but it is more likely to be explicitly mentioned in Level 2 or 3 prompts. This also can explain why using the correctness of codes over one simple prompt is not enough to quantify the difficulty of a task (Finding 2).

7.7 Threats to Validity

Internal validity: One threat to our results' validity comes from how we measure the difficulty of tasks. To mitigate this threat, we combined two metrics to see if a programming task is difficult for LLMs to code using functional and syntactic correctness. We showed it correlates with the *Context Information* level. Those metrics were used in studies investigating LLM capabilities for code generation. We used a linear combination of all levels with the minimum weight for Level 3 (with maximum contextual information) and maximum for Level 1 (with the least information) to emphasize the Level 1 prompt score more. While this might influence the decision on the hard tasks, it is more likely to have some easy tasks be considered as hard, as we showed that the metrics correlate positively with the level. Another threat might come from the generation of the prompts. The transformations used are motivated based on existing works. Moreover, while we have used state-of-the-art GPT-4 to generate them, we manually checked the prompts to ensure that the generative process did not alter semantics. Finally, another validity threat is that we did not consider examples in the prompts. As we mentioned, this can decrease the models' performance and add artificially hard tasks. This was nonetheless necessary to make the comparison fair across tasks, as not all examples contain the same information. However, beyond having some easy tasks

be considered as hard, it does not affect the applicability of *DiffEval*.

External validity: Our sets of tasks from the selected benchmarks could be a threat. The collection of tasks might not be representative of real-world programming tasks to assess LLMs' capabilities. Both HumanEval+ and ClassEval are used to evaluate Code LLM performance [236, 237]. Moreover, the study's objective was to set the scene for assessing task difficulty rather than dwell on the practicalities of the task. So, we believe using tasks from these two benchmarks provides a diverse set of tasks to assess the difficulty of code generation using LLMs.

Reliability validity: We detailed our methodology, and to enable other researchers to reproduce or expand upon our work, we make our code and data publicly available [244].

7.8 Chapter Summary

In this chapter, we aimed to show the limitations of global evaluation regarding LLMs by proposing a novel framework, DiffEval, focusing on assessing the difficulty of code generation tasks for LLMs, identifying hard programming tasks, and creating new hard tasks. We notably show that evaluating a task using several LLMs evaluated on a single prompt can not give a satisfactory answer regarding difficulty assessment. Instead, DiffEval allows this through multiple prompt generation using semantic preserving transformations. This allows a fine-grained analysis of the tasks within a benchmark. By doing so, we highlight that 17% of HumanEval+ tasks and 24% of ClassEval tasks are hard. Such hard tasks are important, as they constitute essential faults as LLMs struggle to solve them despite various inputs. Clustering tasks per topic led to 17 topics for HumanEval+ and 21 for ClassEval. Another advantage of such targeted analysis is that it reuses identified hard tasks to generate more hard tasks and better probes LLMs by targeting their difficulty. In particular, we used 3 of the topics identified per benchmarks, topics with a high number of hard tasks, to further generate 15 new hard tasks per benchmark. Through DiffEval, we thus propose another way to look at benchmarks by considering more local aspects.

CHAPTER 8 CONCLUSION

This thesis aimed to tackle the issue of the effectiveness and trustworthiness of testing techniques applied to DNNs. The focal points are centered around four frameworks, each emphasizing and alleviating one limitation of testing techniques applied to a particular DL paradigm.

To guide our investigation, we defined the high-level research question:

How can we enhance the trustworthiness and effectiveness of testing techniques applied to Deep Learning models to foster users trust in those models given the nature of Deep Learning paradigms?

This high-level question was further refined into four RQs that we addressed in this thesis:

RQ*a*: *How can we reliably detect faults in Deep Reinforcement Learning code?*

This RQ was tackled in Chapter 4 and introduced the framework *RLMutation*, a framework for MT applied to DRL. The objective was to show how to use MT in DRL, redefining the testing approach for the DRL context in terms of the main elements of MT: what mutations are applicable, what detection criteria to use and what constitutes a test case. Notably, we highlight the problem of previous MT approaches, when applied in DL through the effect of stochasticity, showing that previous approaches could misdetect mutations.

We leverage bug taxonomies in DL and DRL to create a list of mutation operators based on existing identified faults in DRL. To prolong the study of mutations in DRL, we also develop HOMs based on listed, more complex mutations. Then, we apply the statistical-based approach described in MT to supervised learning and adapt it to the DRL problem. We define two detection criteria by comparing the distribution of rewards between mutated and non-mutated agents. Finally, we explicitly show the role of the environment as a test case for DRL and that we can effectively generate new environments to constitute more DRL test cases.

Results are based on three different DRL agents (A2C, PPO, and DQN), which were assessed in two environments (CartPole and LunarLander) that are widely used when assessing DRL agents. We show that our defined criteria are better at detecting mutations compared to previous MT definitions by up to 45% and, in particular, leveraging generated environments can further help in catching more mutations, which improves on previous studies. Moreover,

through the study of HOMs in DRL, we show our criteria are capable of detecting HOMs, with more than 50% of HOMs being Subsuming HOMs, which constitute interesting mutations as they highlight the capacity of the generated environments to detect complex mutations. Thus, this study highlights the possibility of applying MT in DRL through a redefinition of the testing technique to the specific DRL context while also showing the benefit of the technique when it comes to identifying issues within DRL specifications.

To enhance the generalizability of the study, future works should aim to expand it using more agents and environments and define more mutations. This last point, in particular, could also help expand the number of HOMs created. Moreover, while effective and straightforward, our environment generation approach could be improved to find more diverse and mutation-revealing environments while preserving the quality of the environments generated. Finally, we showed that the particular patterns of environments detecting a mutation could be used as a possible fault-localization method. So, future works could expand on this idea and integrate it fully.

RQ*b*: *How to account for stochasticity of Mutation Testing in Supervised Learning?*

This RQ was tackled in Chapter 5 and introduced the framework *PMT* (Probabilistic Mutation Testing), a framework to extend MT applied in supervised learning by leveraging Bayesian probability. The objective was to highlight the stochastic issue further when dealing with MT in DL, which we hinted at in the previous chapter, and to find a way to account for it in the MT decision process rather than mitigate it.

The proposed framework leverages Bayesian probability and Bayesian Bagging methodologies to compute the posterior distribution representing the probability of detecting a given mutation. The obtained posterior can be analyzed using summary statistics such as point estimates or credible intervals to make a more informed decision on whether the mutation should be considered detected or not. For a more practical approach, we devised a simple criteria using the obtained posterior by comparing it using probabilistic distance to the two extreme ideal posteriors: the one where the mutation is detected every time and the one where the mutation is never detected. This allows us to give a criterion of detected/not-detected assorted with an effect value, the stronger the effect the more likely the detection/non-detection verdict. We also studied the error of the diverse approximations we made in our framework.

Results are based on three datasets (MNIST, MovieLens, UnityEyes) and models (8-layered Convolutional DNN, Movie Recommender and UnityEyes specific DNN) and nine mutations used in previous works related to MT in DL. We show that the proposed criterion leads to better analysis and gives the user the possibility for a finer-grained analysis, for instance,

by comparing the effect between mutations before making a decision. The criterion allows for more stability and consistency across mutation decisions, with a more informed decision. For instance, 25% of the mutations detected with the state-of-the-art MT definition turn out to have a very uninformative posterior, and so considering this mutation detected was merely based on luck. Looking at the errors in our approximations, we show that the error is negligible for the set of parameters chosen. Moreover, the number of model instances used in our experiments could be decreased to 100 instances, saving computation time while leading to a relatively small error. We finished by discussing the trade-off between PMT and simple MT, showing that PMT can be seen as a generalization of MT and that reducing the number of model instances used in the framework amounts to using MT.

Future works should first expand on the datasets/models and mutations used to enhance the study's generalizability. Moreover, while PMT was applied in a supervised learning setting, the framework is general enough to be adapted to other mutation detection definitions and/or DL paradigms. For instance, applying it in the DRL setting we discussed in the previous Chapter could be interesting. Finally, another avenue could focus on decreasing the number of instances needed while keeping the error of the approximations used manageable.

RQ*c*: *How to transfer test sets across DL models to reduce testing cost while preserving effectiveness?*

This RQ was tackled in Chapter 6 and introduced the framework *GIST* (Generated Inputs Sets Transfer), a framework for the transferability of test sets of DNNs. The objective was to develop the idea of test transferability, borrowed from the transfer learning, to foster an alternative way to applying test generation techniques everytime. The framework should help reduce the computation cost of test generation techniques while preserving the effectiveness delivered by the test sets.

The proposed framework leverages a pool of available DNN models, along with their generated test sets, some similarity metrics defined in the literature to compute the likeness between DNNs as well as a property the user wants the transferred test sets to cover on a new DNN under test, for instance, the faults within the model. The framework works in two phases: In the first phase, the similarity is used to compute the distance between DNNs in a leave-one-out approach. That is, one DNN is, in turn, considered to be the DNN for which we do not have the generated test set and on which we want to transfer some test sets. Using this simulation, the relation between the property of interest and the similarity can be established. In the second phase, we can transfer any number of test sets to cover the chosen property using any new DNN for which we have no generated test set but the validated similarity.

Results are based on two datasets of different modalities (CIFAR-10 for Images and Movie Reviews for Text), with five different DNNs for each modality and three test generation techniques (for Images: a fuzz-based technique and a GAN-based technique, for Text: a semantic/syntactic transformation-based technique). Regarding property, we first focus on fault detection in DNNs using an existing definition of faults in DNNs. However, to show GIST is generalizable to other properties, we also define a property based on the KMNC coverage criterion used in the literature (see Chapter 3). We show that there is a link between certain defined similarity metrics and the property to cover, and this link depends on the test generation technique considered and the property to coverer. Furthermore, those validated similarities can be used to transfer test sets, resulting in better-than-randomized test sets covering properties in all cases. By using transferred test sets, we can recover up to 80% of the property we would have had we generated the test from scratch instead of transferring. Moreover, the transfer paradigm is more effective than reapplying the test generation techniques when considering more than two DNNs to test. This effectiveness is measured by computing the ratio between the proportion of the covered property and the time ratio, the ratio between the computation time of GIST and of reapplying the technique.

To enhance the generalizability of test transfer, even though we strived to cover as many directions as possible, future works should expand on the datasets used, the modalities, and the properties to cover. Moreover, test transfer could be the first step for further amplification, that is, more generation of test sets. Thus, another venue could be to study how to incorporate the test transfer paradigm in the test generation process. Finally, test transfer could be improved by carefully selecting subsets of inputs within each transferred test set. Indeed, currently, transfer is done by transferring the whole test sets, which inevitably increases the number of test inputs considered and causes redundancy in terms of the property to cover, as some inputs will likely cover the same part of the property (e.g., the same fault). Thus, better selection within the test sets could improve the approach's effectiveness.

RQ*d*: *How to assess individual tasks within benchmarks of Large Language Models?*

This RQ was tackled in Chapter 7 and introduced the framework *DiffEval*, a framework for assessing the difficulty of programming tasks in LLMs' benchmarks. The objective was to propose a way to quantify the difficulty of a task in benchmarks. We defined a task as an abstraction of a particular goal (e.g. implement a quicksort), while the prompts are the many ways of expressing the task (e.g. "Write a function to implement a quicksort"). This would allow for a more local and fine-grained analysis, which global metrics such as Accuracy or Pass@k might not offer, and pave the way for a generation of programming tasks with controlled difficulty to supplement new benchmarks for assessments of LLMs.

The proposed framework leverages transformations, namely rephrasing and context information modification, to model the diversity of prompts for a given task. The prompts obtained are then used as inputs to several Code LLMs that have to propose code fragments given the input prompts. Then, using code-related functional and syntactic correctness metrics, each code fragment can be assigned a score that can then be aggregated at the task level to get a difficulty score for the task. Intuitively, we expected the easier tasks to be solved by many LLMs no matter the phrasing or context information given, while harder tasks should be solved using a small proportion of combinations of LLMs/phrasing/context information. Tasks can be analyzed using topic modelling to identify the different task topics. Using those task topics and the identified hard tasks, we can generate new tasks by mutating available hard tasks.

Results are based on two code benchmarks (HumanEval+ and ClassEval) and using five Code LLMs (DeepSeek, MagiCoder, CodeLLama, Gemma and GPT3.5). We first verify the usefulness of the context information by showing that the code fragments score increases as the amount of context information increases. Then, we compare tasks labelled as hard using a simple ensemble of LLMs using greedy decoding compared to the DiffEval approach, and we show that our identified hard tasks are indeed hard with the ensemble approach, but the contrary is not true. This highlights that using a simple ensemble methodology to judge task difficulty is insufficient. We then analyze the difficulty of tasks within the two benchmarks under test, showing that, while the two benchmarks lead to similar Accuracy-wise scores at equivalent models, they do not necessarily have the same task difficulty scores. For instance, 17% of HumanEval+ tasks are hard, and 24% of ClassEval tasks are hard. Hard tasks within those benchmarks are not focused on some topics but are somewhat scattered across topics. Using 6 of those topics and hard tasks, we can generate 15 new tasks per benchmark that are also scored as hard using our approach, showing a possible way for generating new tasks to obtain a targeted benchmark of hard tasks.

To enhance the study, more benchmarks should be analyzed. Moreover, while we focused on the code generation paradigm, the approach could be used in other paradigms such as code summarization, Question/Answering, etc. This would allow for a better analysis of the difficulty of the tasks of existing benchmarks. Another avenue of research would be to improve the generation of new hard tasks to generate tasks with a controlled level of difficulty to match it to a particular capacity of an LLM under test to improve testing ability.

8.1 Limitations

While aiming to tackle multiple points, the thesis could not cover all particularity of testing in DL. For instance, among the limitations of testing pointed out in Chapter 2.3.4 and Chapter 3.4, we did not tackle certain ones such as numerical instability, hyperparameter selection of testing techniques or realism of generated test inputs. Moreover, while we try to explore the limitations of testing techniques by accounting for several DL paradigms, we could not account for all. Additionally, while testing techniques and their limitations tackled in this thesis should still be a concern in the near future, new limitations and new testing techniques could be introduced with the rise of new paradigms in DL. For instance, the introduction of LLMs already introduced new failures/faults that previous DNNs architecture would not exhibit. As such, analyzing and solving the limitations of testing techniques in DL is a constant endeavour. Beyond those general concerns, here are a few additional points that could affect the results obtained in our studies:

1. *Diversity of models and datsets:* To demonstrate the performance of our framework, we strived everytime to use different models and datasets within the scope of the study. While the results achieved with those models and datasets are encouraging, more evaluations are needed with additional models and datasets to cement the generalizability of our proposed frameworks. The particularity of each framework (e.g. mutation operators, property to cover, etc.) should also be expanded.

2. *DL Paradigms generalization:* All of our frameworks were studied within the scope of a precise DL paradigm. While we define the frameworks as general as possible, they should be expanded to other paradigms to improve their generalizability further.

3. *Cost of the analysis:* All of our frameworks generally rely on multiple available DNNs and/or multiple test inputs. While using multiple DNNs or inputs is paramount to account for the effect of DL nature when testing them, notably with the stochasticity, this incurs an additional cost on the testing techniques. We account for this cost mostly in Chapter 6 with the GIST framework for test transfer, yet this framework still posits we have access to some DNNs before applying the framework. Thus, our framework applications are limited when it comes to very computationally intensive tasks.

8.2 Future Research

In light of this thesis, two things are clear: testing and existing techniques are a cornerstone of fostering DNNs' trustworthiness by detecting faults. However, there is a long road ahead to

ensure that such techniques adequately test DNNs from the perspective of their effectiveness and trustworthiness. The proposed frameworks in this thesis are but one step in this direction, and we hope those studies motivate the research community to expand on them. Based on our observations, we describe possible future research avenues.

1. *Expanding generalizability of the proposed frameworks:* One main avenue of improvement is first to develop the proposed frameworks. For instance, RLMutation could also be applied to multi-agent problems instead of single agents, implying a potential modification of the MT definition we introduced. Moreover, the proposed frameworks could expanded as well to other DL paradigms or tasks. For instance, the framework GIST can be applied to code-based LLMs or DiffEval to hard environments for code summarization or Question/Answering.

2. *Reducing instances/models needed to apply frameworks:* As highlighted in the limitations, the proposed frameworks require many inputs/models. This can be a limiting factor in a constrained context. Thus, future works could tackle this issue by finding a methodology to reduce the number of inputs/models needed for the frameworks to operate. A trade-off between the required number and the potential impact on the trustworthiness (e.g., error in approximations) should be considered, as we discussed in the framework PMT (see Chapter 5)

3. *Investigating overlap of faults between models:* Testing techniques for faults at the model-level generally focus on finding them for a particular DNN. It's, for instance, the case of the multiple testing techniques we evaluated in GIST (see Chapter 6) or that we described in Chapter 3. However, relations between faults across DNNs of the same class (for instance, all Convolutional DNNs) are rarely studied, which limits the relevance of the limits when it comes to finding faults within DNNs. Indeed, while it is interesting to find faults in a DNN under test, it could be beneficial to understand if said faults are specific to a DNN and the results of the learning process or if they are particularity due to the DNN architecture or task that makes any DNNs in this class fail. We are not aware of such studies in image-based DNNs. Studies generally focus on testing DNNs in isolations for faults, eventually checking for transferability between them as an artifact [158, 159]. Similarly, we showed in Chapter 6 that test transfer between DNNs, showing faults between them are linked. However, this is limited in terms of proper analysis, partly because failures in image-based models might be more challenging to interpret. In the case of LLMs, more studies have started to analyze the evaluation results between them. For instance, we studied [246] the different types of

mistakes that code LLMs make by taxonomizing them, showing some common mistakes but simultaneously highlighting the differences. Liu et al. [272] with their CodeMind framework, shows the struggle of LLMs on code constructs and highlights differences and likenesses between LLMs. In Chapter 7, we analyzed how LLMs quantify the difficulty of tasks across different code LLMs and highlighted as difficult the tasks that generally resulted in most LLMs failing on them no matter the formulation of the prompts. This aspect is crucial for benchmarking efforts, as it can help better quantify what the next benchmarks should be by focusing on the precise limitations of LLMs rather than relying on a global score over the benchmark.

4. *Leveraging observations made in the frameworks to improve testing techniques:* Beyond evaluating testing techniques, while the frameworks we proposed were mainly focused on improving the effectiveness and trustworthiness of testing techniques in DL, we made some interesting peripheral observations in many of them which could be leveraged for other avenues. In Chapter 4, we saw that the particular environments detecting or not the mutations outlined some patterns that could be used potentially in a fault-localization objective, that is, finding where the fault would be in the code. Using mutants for fault-localization was already addressed in traditional programs [273]. So, we believe a similar approach could be leveraged for DL with the added constraint of accounting for the stochasticity, as we showed. In Chapter 6, we studied the possibility of transferring test sets with GIST. Yet, an interesting improvement would be extending the framework to see how test transferability could be used to amplify test input generation. Starting from the transferred test sets, how can the gap be bridged in terms of the part of the property that is not covered, for example, faults not detected by the transferred test sets. Finally, in Chapter 7, DiffEval showed a way of quantifying task difficulty in code LLMs. Moreover, we showed it was possible to generate new hard tasks using available ones. While preliminary, this shows that the automatic generation of benchmarks with targeted objectives could be doable. For instance, Zhuang et al. [274] proposed an adaptive testing methodology that automatically adjusts questions' characteristics from a pool of questions, such as difficulty. It would be interesting to see if a similar approach could be used to generate tailored benchmarks instead, to properly evaluate code LLMs capacity.

REFERENCES

[1] "Convolutional neural networks for text classification," https://www.davidsbatista.net/blog/2018/03/31/SentenceClassificationConvNets/, 2018.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[3] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[4] K. Haan, "24 top ai statistics and trends in 2024," *Forbes*. [Online]. Available: https://www.forbes.com/advisor/business/ai-statistics/

[5] "Gpt-3.5 turbo," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2023.

[6] "Copilot," https://copilot.microsoft.com/, 2024.

[7] G. staff and agencies, "Tesla autopilot feature was involved in 13 fatal crashes, us regulator says," *The Guardian*. [Online]. Available: https://www.theguardian.com/technology/2024/apr/26/tesla-autopilot-fatal-crash

[8] O. Rainio, J. Teuho, and R. Klén, "Evaluation metrics and statistical tests for machine learning," *Scientific Reports*, vol. 14, no. 1, p. 6086, 2024.

[9] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[10] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[11] 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*, ser. SOSP '17. ACM, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1145/3132747.3132785

[12] J. Chen, J. Wang, X. Ma, Y. Sun, J. Sun, P. Zhang, and P. Cheng, "Quote: Quality-oriented testing for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: https://doi.org/10.1145/3582573

[13] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empirical Software Engineering*, vol. 25, pp. 5193–5254, 2020.

[14] H. Ben 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)*, 2019, pp. 454–458.

[15] Y. Sun, D. Lao, G. Sundaramoorthi, and A. Yezzi, "Surprising instabilities in training deep networks and a theoretical analysis," *Advances in Neural Information Processing Systems*, vol. 35, pp. 19 567–19 578, 2022.

[16] M. G. Altarabichi, S. Nowaczyk, S. Pashami, P. S. Mashhadi, and J. Handl, "Rolling the dice for better deep learning performance: A study of randomness techniques in deep neural networks," *Information Sciences*, vol. 667, p. 120500, 2024.

[17] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 74–84.

[18] Z. Wang, S. Xu, L. Fan, X. Cai, L. Li, and Z. Liu, "Can coverage criteria guide failure discovery for image classifiers? an empirical study," *ACM Trans. Softw. Eng. Methodol.*, jun 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3672446

[19] F. Tambon, F. Khomh, and G. Antoniol, "A probabilistic framework for mutation testing in deep neural networks," *Information and Software Technology*, vol. 155, p. 107129, 2023.

[20] F. Tambon, V. Majdinasab, A. Nikanjam, F. Khomh, and G. Antoniol, "Mutation testing of deep reinforcement learning based on real faults," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 188–198.

[21] F. Tambon, F. Khomh, and G. Antoniol, "Gist: Generated inputs sets transferability in deep learning," *ACM Transactions on Software Engineering and Methodology*, 2023.

[22] F. Tambon, A. Nikanjam, F. Khomh, and G. Antoniol, "Assessing programming task difficulty for efficient evaluation of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21227

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[24] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 315–323. [Online]. Available: https://proceedings.mlr.press/v15/glorot11a.html

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[26] Zhou and Chellappa, "Computation of optical flow using a neural network," in *IEEE 1988 international conference on neural networks*. IEEE, 1988, pp. 71–78.

[27] D. Rothman, *Transformers for Natural Language Processing and Computer Vision - Third Edition*. Packt, 2024.

[28] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong *et al.*, "Understanding llms: A comprehensive overview from training to inference," *arXiv preprint arXiv:2401.02038*, 2024.

[29] J. Alammar, "The illustrated transformer," https://jalammar.github.io/illustrated-transformer/, accessed: 2024-07-07.

[30] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022.

[31] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[32] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[33] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, 2nd ed. Wiley Publishing, 2018.

[34] I. Jovanović, "Software testing methods and techniques," *The IPSI BgD transactions on internet research*, vol. 30.

[35] E. W. Dijkstra, "Notes on structured programming," 1970.

[36] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571730

[37] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow," *Empirical Software Engineering*, vol. 29, no. 1, p. 10, 2024.

[38] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[39] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[40] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[41] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[42] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[43] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[44] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[45] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018, pp. 1–6.

[46] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[47] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[48] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, pp. 273–297, 1995.

[49] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[50] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, sep 2018. [Online]. Available: https://doi.org/10.1145/3234150

[51] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013721000198

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer, 2016, pp. 630–645.

[55] 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.

[56] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[57] 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*, 2009, pp. 248–255.

[58] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016. [Online]. Available: https://arxiv.org/abs/1506.02640

[59] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[60] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[61] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[62] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[63] M. Denil, A. Demiraj, N. Kalchbrenner, P. Blunsom, and N. de Freitas, "Modelling, visualising and summarising documents with a single convolutional neural network," *arXiv preprint arXiv:1406.3830*, 2014.

[64] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[65] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

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

[67] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[68] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing [review article]," *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[69] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[70] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 684–10 695.

[71] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[72] 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.

[73] 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.

[74] 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.

[75] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: a taxonomy and a detection approach," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–32, 2022.

[76] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[77] ——, "An empirical study on bugs inside tensorflow," in *International Conference on Database Systems for Advanced Applications*. Springer, 2020, pp. 604–620.

[78] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Trans. Softw. Eng. Methodol.*, mar 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3587155

[79] A. D'Amour, K. Heller, D. Moldovan, B. Adlam, B. Alipanahi, A. Beutel, C. Chen, J. Deaton, J. Eisenstein, M. D. Hoffman *et al.*, "Underspecification presents challenges for credibility in modern machine learning," *Journal of Machine Learning Research*, vol. 23, no. 226, pp. 1–61, 2022.

[80] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *CVPR 2011*, 2011, pp. 1521–1528.

[81] M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz, "Invariant risk minimization," 2020. [Online]. Available: https://arxiv.org/abs/1907.02893

[82] 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.

[83] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/e2c420d928d4bf8ce0ff2ec19b371514-Paper.pdf

[84] I. Dunn, H. Pouget, D. Kroening, and T. Melham, "Exposing previously undetectable faults in deep neural networks," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 56–66. [Online]. Available: https://doi.org/10.1145/3460319.3464801

[85] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "DLFuzz: differential fuzzing testing of deep learning systems," *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Oct 2018.

[86] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 131–138.

[87] N. Dziri, A. Madotto, O. Zaïane, and A. J. Bose, "Neural path hunter: Reducing hallucination in dialogue systems via path grounding," *arXiv preprint arXiv:2104.08455*, 2021.

[88] J. Wang, Y. Zhou, G. Xu, P. Shi, C. Zhao, H. Xu, Q. Ye, M. Yan, J. Zhang, J. Zhu, J. Sang, and H. Tang, "Evaluation and analysis of hallucination in large vision-language models," 2023. [Online]. Available: https://arxiv.org/abs/2308.15126

[89] F. Tambon, G. Laberge, L. An, A. Nikanjam, P. S. N. Mindom, Y. Pequignot, F. Khomh, G. Antoniol, E. Merlo, and F. Laviolette, "How to certify machine learning based safety-critical systems? a systematic literature review," *Automated Software Engineering*, vol. 29, no. 2, p. 38, 2022.

[90] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020.

[91] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2020.

[92] J. Terven, D. M. Cordova-Esparza, A. Ramirez-Pedraza, and E. A. Chavez-Urbiola, "Loss functions and metrics in deep learning," 2023. [Online]. Available: https://arxiv.org/abs/2307.02694

[93] M. Naser and A. Alavi, "Insights into performance fitness and error metrics for machine learning," *arXiv preprint arXiv:2006.00887*, 2020.

[94] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, mar 2024. [Online]. Available: https://doi.org/10.1145/3641289

[95] A. B. Sai, A. K. Mohankumar, and M. M. Khapra, "A survey of evaluation metrics used for nlg systems," *ACM Comput. Surv.*, vol. 55, no. 2, jan 2022. [Online]. Available: https://doi.org/10.1145/3485766

[96] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *Advances in neural information processing systems*, vol. 29, 2016.

[97] S. Banerjee, A. Lavie *et al.*, "An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the ACL-2005 Workshop on Intrinsic and Extrinsic Evaluation Measures for MT and/or Summarization*, 2005, pp. 65–72.

[98] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[99] N. Mathur, T. Baldwin, and T. Cohn, "Putting evaluation in context: Contextual embeddings improve machine translation evaluation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2799–2808.

[100] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[101] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 120–131. [Online]. Available: https://doi.org/10.1145/3238147.3238202

[102] 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*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, p. 146–157.

[103] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "DeepCT: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 614–618.

[104] J. Sekhon and C. Fleming, "Towards improved testing for deep learning," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019, pp. 85–88.

[105] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.

[106] 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.

[107] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*,

ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 177–188. [Online]. Available: https://doi.org/10.1145/3395363.3397357

[108] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 297–298.

[109] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2012.

[110] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic finite state machines," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 141–150.

[111] N. Chetouane, L. Klampfl, and F. Wotawa, "Investigating the effectiveness of mutation testing tools in the context of deep neural networks," in *Advances in Computational Intelligence*, I. Rojas, G. Joya, and A. Catala, Eds. Cham: Springer International Publishing, 2019, pp. 766–777.

[112] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," 2018. [Online]. Available: https://arxiv.org/abs/1805.05206

[113] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1158–1161.

[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)*, 2018, pp. 108–115.

[115] Y. Lu, W. Sun, and M. Sun, "Towards mutation testing of reinforcement learning systems," *Journal of Systems Architecture*, vol. 131, p. 102701, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122001977

[116] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: mutation testing of deep learning systems based on real faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 67–78.

[117] A. Panichella and C. C. S. Liem, *What Are We Really Testing in Mutation Testing for Machine Learning? A Critical Reflection.* IEEE Press, 2021, p. 66–70. [Online]. Available: https://doi.org/10.1109/ICSE-NIER52604.2021.00022

[118] L. Gauerhof, R. Hawkins, C. Picardi, C. Paterson, Y. Hagiwara, and I. Habli, "Assuring the safety of machine learning for pedestrian detection at crossings," in *Computer Safety, Reliability, and Security*, A. Casimiro, F. Ortmeier, F. Bitsch, and P. Ferreira, Eds. Cham: Springer International Publishing, 2020, pp. 197–212.

[119] R. Salay and K. Czarnecki, "Using machine learning safely in automotive software: An assessment and adaption of software process requirements in iso 26262," *ArXiv*, vol. abs/1808.01614, 2018.

[120] W. Shen, Y. Li, Y. Han, L. Chen, D. Wu, Y. Zhou, and B. Xu, "Boundary sampling to boost mutation testing for deep learning models," *Information and Software Technology*, vol. 130, p. 106413, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301737

[121] E. Grefenstette, R. Stanforth, B. O'Donoghue, J. Uesato, G. Swirszcz, and P. Kohli, "Strength in numbers: Trading-off robustness and computation via adversarially-trained ensembles," *CoRR*, vol. abs/1811.09300, 2018. [Online]. Available: http://arxiv.org/abs/1811.09300

[122] H. Xu, Z. Chen, W. Wu, Z. Jin, S. Kuo, and M. Lyu, "Nv-dnn: Towards fault-tolerant dnn systems with n-version programming," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2019, pp. 44–47.

[123] F. Machida, "N-version machine learning models for safety critical systems," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2019, pp. 48–51.

[124] Y. Li, Y. Liu, M. Li, Y. Tian, B. Luo, and Q. Xu, "D2NN: A fine-grained dual modular redundancy framework for deep neural networks," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19).* New York, NY, USA: ACM, 2019, p. 138–147.

[125] C. Cheng, G. Nührenberg, and H. Yasuoka, "Runtime monitoring neuron activation patterns," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 300–303.

[126] M. S. Ramanagopal, C. Anderson, R. Vasudevan, and M. Johnson-Roberson, "Failing to learn: Autonomously identifying perception failures for self-driving cars," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, p. 3860–3867, Oct 2018.

[127] 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.

[128] X. Xie, Z. Zhang, T. Y. Chen, Y. Liu, P.-L. Poon, and B. Xu, "Mettle: A metamorphic testing approach to assessing and validating unsupervised machine learning systems," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1293–1322, 2020.

[129] S. Hyun, M. Guo, and M. A. Babar, "Metal: Metamorphic testing framework for analyzing large-language model qualities," *arXiv preprint arXiv:2312.06056*, 2023.

[130] S. Demir, H. F. Eniser, and A. Sen, "Deepsmartfuzzer: Reward guided test generation for deep learning," 2019, arXiv preprint arXiv:arXiv: 1911.10621.

[131] 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*, ser. ICSE '18. New York, NY, USA: ACM, 2018, p. 303–314.

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

[133] P. Zhang, Q. Dai, and S. Ji, "Condition-guided adversarial generative testing for deep learning systems," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019, pp. 71–72.

[134] M. Yan, L. Wang, and A. Fei, "ARTDL: Adaptive random testing for deep learning systems," *IEEE Access*, vol. 8, pp. 3055–3064, 2020.

[135] T. Dreossi, S. Ghosh, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Systematic testing of convolutional neural networks for autonomous driving," 2017, arXiv preprint arXiv:1708.03309.

[136] S. Udeshi, X. Jiang, and S. Chattopadhyay, "Callisto: Entropy-based test generation and data quality assessment for machine learning systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 448–453.

[137] K. Pei, Y. Cao, J. Yang, and S. Jana, "Towards practical verification of machine learning: The case of computer vision systems," 2017, arXiv preprint arXiv:1712.01785.

[138] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 408–426.

[139] S. Yaghoubi and G. Fainekos, "Gray-box adversarial testing for control systems with machine learning components," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: ACM, 2019, pp. 179—-184.

[140] J. Morris, E. Lifland, J. Lanchantin, Y. Ji, and Y. Qi, "Reevaluating adversarial examples in natural language," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 3829–3839. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.341

[141] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, "Is bert really robust? a strong baseline for natural language attack on text classification and entailment," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 8018–8025.

[142] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, p. 318–328.

[143] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1555–1562.

[144] C. Wolschke, T. Kuhn, D. Rombach, and P. Liggesmeyer, "Observation based creation of minimal test suites for autonomous vehicles," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 294–301.

[145] I. Gur, N. Jaques, Y. Miao, J. Choi, M. Tiwari, H. Lee, and A. Faust, "Environment generation for zero-shot compositional reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4157–4169, 2021.

[146] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging procedural generation to benchmark reinforcement learning," in *International conference on machine learning*. PMLR, 2020, pp. 2048–2056.

[147] M. Biagiola and P. Tonella, "Testing the plasticity of reinforcement learning-based systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1145/3511701

[148] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[149] D. Müller, I. Soto-Rey, and F. Kramer, "Towards a guideline for evaluation metrics in medical image segmentation," *BMC Research Notes*, vol. 15, no. 1, p. 210, 2022.

[150] T. Hu and X.-H. Zhou, "Unveiling llm evaluation focused on metrics: Challenges and solutions," 2024. [Online]. Available: https://arxiv.org/abs/2404.09135

[151] A. Reinke, M. D. Tizabi, C. H. Sudre, M. Eisenmann, T. Rädsch, M. Baumgartner, L. Acion, M. Antonelli, T. Arbel, S. Bakas, P. Bankhead, A. Benis, M. Blaschko, F. Buettner, M. J. Cardoso, J. Chen, V. Cheplygina, E. Christodoulou, B. Cimini, G. S. Collins, S. Engelhardt, K. Farahani, L. Ferrer, A. Galdran, B. van Ginneken, B. Glocker, P. Godau, R. Haase, F. Hamprecht, D. A. Hashimoto, D. Heckmann-Nötzel, P. Hirsch, M. M. Hoffman, M. Huisman, F. Isensee, P. Jannin, C. E. Kahn, D. Kainmueller, B. Kainz, A. Karargyris, A. Karthikesalingam, A. E. Kavur, H. Kenngott, J. Kleesiek, A. Kleppe, S. Kohler, F. Kofler, A. Kopp-Schneider, T. Kooi, M. Kozubek, A. Kreshuk, T. Kurc, B. A. Landman, G. Litjens, A. Madani, K. Maier-Hein, A. L. Martel, P. Mattson, E. Meijering, B. Menze, D. Moher, K. G. M. Moons, H. Müller, B. Nichyporuk, F. Nickel, M. A. Noyan, J. Petersen, G. Polat, S. M. Rafelski, N. Rajpoot, M. Reyes, N. Rieke, M. Riegler, H. Rivaz, J. Saez-Rodriguez, C. I. Sánchez, J. Schroeter, A. Saha, M. A. Selver, L. Sharan, S. Shetty, M. van Smeden, B. Stieltjes, R. M. Summers, A. A. Taha, A. Tiulpin, S. A. Tsaftaris, B. V. Calster, G. Varoquaux, M. Wiesenfarth, Z. R. Yaniv, P. Jäger, and L. Maier-Hein, "Common limitations of image processing metrics: A picture story," 2023. [Online]. Available: https://arxiv.org/abs/2104.05642

[152] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim, "Is neuron coverage a meaningful measure for testing deep neural networks?" in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium*

on the Foundations of Software Engineering, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 851–862. [Online]. Available: https://doi.org/10.1145/3368089.3409754

[153] B. Chen, M. Wen, Y. Shi, D. Lin, G. K. Rajbahadur, and Z. M. Jiang, "Towards training reproducible deep learning models," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2202–2214.

[154] N. A. Lynnerup, L. Nolling, R. Hasle, and J. Hallam, "A survey on reproducibility by evaluating deep reinforcement learning algorithms on real-world robots," in *Conference on Robot Learning*. PMLR, 2020, pp. 466–489.

[155] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 1–10.

[156] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2014. [Online]. Available: https://arxiv.org/abs/1312.6199

[157] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and S. Ramesh, "A search-based testing approach for deep reinforcement learning agents," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3715–3735, 2023.

[158] K. Liang, J. Y. Zhang, B. Wang, Z. Yang, S. Koyejo, and B. Li, "Uncovering the connections between adversarial transferability and knowledge transferability," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6577–6587.

[159] L. Wu and Z. Zhu, "Towards understanding and improving the transferability of adversarial examples in deep neural networks," in *Asian Conference on Machine Learning*. PMLR, 2020, pp. 837–850.

[160] B. Yan, J. Zhang, Z. Yuan, S. Shan, and X. Chen, "Evaluating the quality of hallucination benchmarks for large vision-language models," 2024. [Online]. Available: https://arxiv.org/abs/2406.17115

[161] H. Zhang, J. Zhang, and X. Wan, "Quantity matters: Towards assessing and mitigating number hallucination in large vision-language models," 2024. [Online]. Available: https://arxiv.org/abs/2403.01373

[162] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, and B. Xiang, "Recode: Robustness evaluation of code generation models," 2022.

[163] M. Mizrahi, G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky, "State of what art? a call for multi-prompt llm evaluation," 2024.

[164] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 251–262.

[165] H. Ben Braiek and F. Khomh, "Testing feedforward neural networks training programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3529318

[166] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "Deepfd: automated fault diagnosis and localization for deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 573–585. [Online]. Available: https://doi.org/10.1145/3510003.3510099

[167] Y. Zhi, X. Xie, C. Shen, J. Sun, X. Zhang, and X. Guan, "Seed selection for testing deep neural networks," *ACM Trans. Softw. Eng. Methodol.*, jul 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3607190

[168] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, jan 2021. [Online]. Available: https://doi.org/10.1145/3417330

[169] Q. Hu, Y. Guo, M. Cordy, X. Xie, L. Ma, M. Papadakis, and Y. Le Traon, "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1145/3511598

[170] Z. Aghababaeyan, M. Abdellatif, L. Briand, R. S, and M. Bagherzadeh, "Black-box testing of deep neural networks through test case diversity," *IEEE Transactions on Software Engineering*, vol. 49, no. 05, pp. 3182–3204, may 2023.

[171] Y. Lu, W. Sun, and M. Sun, "Towards mutation testing of reinforcement learning systems," *Journal of Systems Architecture*, vol. 131, p. 102701, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122001977

[172] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[173] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. G. Bellemare, "Deep reinforcement learning at the edge of the statistical precipice," in *NeurIPS*, 2021.

[174] J. A. Nelder and R. W. M. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, pp. 370–384, 1972. [Online]. Available: http://www.jstor.org/stable/2344614

[175] K. Kelley and K. J. Preacher, "On effect size," *Psychological Method*, vol. 17, no. 2, pp. 137–152, 2012.

[176] H. Cramer, *Mathematical Methods of Statistics*. Princeton University Press, 1946.

[177] "Replication Package RLMutation," https://github.com/FlowSs/RLMutation, 2022.

[178] T. Everitt, V. Krakovna, L. Orseau, M. Hutter, and S. Legg, "Reinforcement learning with a corrupted reward channel," 2017. [Online]. Available: https://arxiv.org/abs/1705.08417

[179] J. Romoff, P. Henderson, A. Piche, V. Francois-Lavet, and J. Pineau, "Reward estimation for variance reduction in deep reinforcement learning," in *Proceedings of The 2nd Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., vol. 87. PMLR, 29–31 Oct 2018, pp. 674–699. [Online]. Available: https://proceedings.mlr.press/v87/romoff18a.html

[180] "Cart Pole," https://www.gymlibrary.dev/environments/classic_control/cart_pole/, 2022.

[181] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[182] "Lunar Lander," https://www.gymlibrary.dev/environments/box2d/lunar_lander/, 2022.

[183] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[184] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao, "Research progress of flaky tests," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 639–646.

[185] J. Kerman, "Neutral noninformative and informative conjugate beta and gamma prior distributions," *Electronic Journal of Statistics*, vol. 5, no. none, pp. 1450 – 1470, 2011. [Online]. Available: https://doi.org/10.1214/11-EJS648

[186] P. Bühlmann, "Discussion of big bayes stories and bayesbag," *Statistical Science*, vol. 29, no. 1, pp. 91–94, 2014. [Online]. Available: http://www.jstor.org/stable/43288454

[187] B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *The Annals of Statistics*, vol. 7, no. 1, pp. 1 – 26, 1979. [Online]. Available: https://doi.org/10.1214/aos/1176344552

[188] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[189] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

[190] J. H. Huggins and J. W. Miller, "Robust inference and model criticism using bagged posteriors," 2019. [Online]. Available: https://arxiv.org/abs/1912.07104

[191] C. L. M. Hespanhol L., Vallio C. S. and S. B. T., "Understanding and interpreting confidence and credible intervals around effect estimates," *Brazilian journal of physical therapy*, vol. 23, no. 4, pp. 290–301, 2019. [Online]. Available: https://doi.org/10.1016/j.bjpt.2018.12.006

[192] S. S. Sawilowsky, "New effect size rules of thumb," *Journal of modern applied statistical methods*, vol. 8, no. 2, p. 26, 2009.

[193] S. H. E. Koehler, E. Brown, "On the assessment of monte carlo error in simulation-based statistical analyses," *The American statistician*, vol. 63, no. 2, pp. 155–162, 2009.

[194] B. Efron, "Jackknife-after-bootstrap standard errors and influence functions," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 54, no. 1, pp. 83–111, 1992.

[195] K. M. C. Model, 2022, available at https://keras.io/examples/vision/mnist_convnet/.

[196] M. R. Dataset, 2022, available at http://files.grouplens.org/datasets/movielens/ml-latest-small.zip.

[197] K. M. R. Dataset, 2022, available at https://keras.io/examples/structured_data/collaborative_filtering_movielens/.

[198] E. Wood, T. Baltrušaitis, L.-P. Morency, P. Robinson, and A. Bulling, "Learning an appearance-based gaze estimator from one million synthesised images," in *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research &; Applications*, ser. ETRA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 131–138. [Online]. Available: https://doi.org/10.1145/2857491.2857492

[199] A. implementation of a multimodal CNN for appearance-based gaze estimation, 2022, available at https://github.com/dlsuroviki/UnityEyesModel.

[200] F. Tambon, "Replication package pmut," 2022, available at https://github.com/FlowSs/PMT and https://zenodo.org/record/7325042.

[201] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[202] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering." *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[203] D. Moulavi, P. A. Jaskowiak, R. J. Campello, A. Zimek, and J. Sander, "Density-based clustering validation," in *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 2014, pp. 839–847.

[204] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042787901257

[205] E. Boix-Adsera, H. Lawrence, G. Stepaniants, and P. Rigollet, "Gulp: a prediction-based metric between representations," *Advances in Neural Information Processing Systems*, vol. 35, pp. 7115–7127, 2022.

[206] Y. Li, Z. Zhang, B. Liu, Z. Yang, and Y. Liu, "Modeldiff: Testing-based dnn similarity comparison for model reuse detection," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 139–151.

[207] M. Klabunde, T. Schumacher, M. Strohmaier, and F. Lemmerich, "Similarity of neural network models: A survey of functional and representational measures," 2023.

[208] F. Ding, J.-S. Denain, and J. Steinhardt, "Grounding representation similarity through statistical testing," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 1556–1568. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/0c0bf917c7942b5a08df71f9da626f97-Paper.pdf

[209] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," in *International conference on machine learning*. PMLR, 2019, pp. 3519–3529.

[210] A. Morcos, M. Raghu, and S. Bengio, "Insights on representational similarity in neural networks with canonical correlation," *Advances in neural information processing systems*, vol. 31, 2018.

[211] O. Madani, D. Pennock, and G. Flake, "Co-validation: Using model disagreement on unlabeled data to validate classification algorithms," *Advances in neural information processing systems*, vol. 17, 2004.

[212] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79 – 86, 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729694

[213] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *City*, vol. 1, no. 2, p. 1, 2007.

[214] W. Li, "Cifar-zoo: Pytorch implementation of cnns for cifar dataset," https://github.com/BIGBALLON/CIFAR-ZOO, 2019.

[215] B. Pang and L. Lee, "Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales," in *Proceedings of the ACL*, 2005.

[216] "Huggingface," https://huggingface.co/, 2023.

[217] "Replication package gist," https://github.com/FlowSs/GIST or https://zenodo.org/records/10028594, 2023.

[218] V. Riccio and P. Tonella, "When and why test generators for deep learning produce invalid inputs: An empirical study," in *Proceedings of the 45th International*

Conference on Software Engineering, ser. ICSE '23. IEEE Press, 2023, p. 1161–1173. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00104

[219] 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)*, 2019, pp. 89–92.

[220] M. Contributors, "MMGeneration: Openmmlab generative model toolbox and benchmark," https://github.com/open-mmlab/mmgeneration, 2021.

[221] J. Morris, E. Lifland, J. Y. Yoo, J. Grigsby, D. Jin, and Y. Qi, "Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 119–126.

[222] J. Singh, B. McCann, R. Socher, and C. Xiong, "BERT is not an interlingua and the bias of tokenization," in *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 47–55. [Online]. Available: https://aclanthology.org/D19-6106

[223] W. Shen, Y. Li, L. Chen, Y. Han, Y. Zhou, and B. Xu, "Multiple-boundary clustering and prioritization to promote neural network retraining," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 410–422. [Online]. Available: https://doi.org/10.1145/3324884.3416621

[224] M. G. KENDALL, "A NEW MEASURE OF RANK CORRELATION," *Biometrika*, vol. 30, no. 1-2, pp. 81–93, 06 1938. [Online]. Available: https://doi.org/10.1093/biomet/30.1-2.81

[225] R. A. Armstrong, "When to use the bonferroni correction," *Ophthalmic and Physiological Optics*, vol. 34, no. 5, pp. 502–508, 2014.

[226] D. Li, R. Jin, J. Gao, and Z. Liu, "On sampling top-k recommendation evaluation," ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2114–2124. [Online]. Available: https://doi.org/10.1145/3394486.3403262

[227] R. S. King, *Cluster Analysis and Data Mining: An Introduction*. Dulles, VA, USA: Mercury Learning & Information, 2014.

[228] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *arXiv preprint arXiv:2002.04803*, 2020.

[229] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Faster or slower? performance mystery of python idioms unveiled with empirical evidence," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1495–1507.

[230] A. Csiszárik, P. Kőrösi-Szabó, A. Matszangosz, G. Papp, and D. Varga, "Similarity and matching of neural network representations," *Advances in Neural Information Processing Systems*, vol. 34, pp. 5656–5668, 2021.

[231] M. Kaya and H. Ş. Bilge, "Deep metric learning: A survey," *Symmetry*, vol. 11, no. 9, p. 1066, 2019.

[232] 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*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 146–157. [Online]. Available: https://doi.org/10.1145/3293882.3330579

[233] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, jan 2021. [Online]. Available: https://doi.org/10.1145/3417330

[234] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 1–10.

[235] M. Attaoui, H. Fahmy, F. Pastore, and L. Briand, "Black-box safety analysis and retraining of dnns based on feature extraction and clustering," vol. 32, no. 3, apr 2023. [Online]. Available: https://doi.org/10.1145/3550271

[236] "Evalplus leaderboard," https://evalplus.github.io/leaderboard.html, 2024.

[237] "Classeval leaderboard," https://fudanselab-classeval.github.io/leaderboard.html, 2024.

[238] C. Shi, H. Yang, D. Cai, Z. Zhang, Y. Wang, Y. Yang, and W. Lam, "A thorough examination of decoding methods in the era of llms," *arXiv preprint arXiv:2402.06925*, 2024.

[239] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[240] M. L. Siddiq, S. Dristi, J. Saha, and J. C. S. Santos, "Quality assessment of prompts used in code generation," 2024.

[241] L. Weber, E. Bruni, and D. Hupkes, "Mind the instructions: a holistic evaluation of consistency and interactions in prompt-based learning," 2023.

[242] H. Gonen, S. Iyer, T. Blevins, N. A. Smith, and L. Zettlemoyer, "Demystifying prompts in language models via perplexity estimation," *arXiv preprint arXiv:2212.04037*, 2022.

[243] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[244] "Replication package diffeval," https://anonymous.4open.science/r/anonymous_repository, 2024.

[245] C. Liu, S. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Can large language models reason about code?"

[246] F. Tambon, A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," 2024.

[247] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

[248] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[249] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2149–2160.

[250] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar *et al.*, "Starcoder 2 and the stack v2: The next generation," 2024.

[251] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[252] M. Grootendorst, "Bertopic: Neural topic modeling with a class-based tf-idf procedure," *arXiv preprint arXiv:2203.05794*, 2022.

[253] P. Hämäläinen, M. Tavast, and A. Kunnari, "Evaluating large language models in generating synthetic hci research data: a case study," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3580688

[254] T. Fütterer, C. Fischer, A. Alekseeva, X. Chen, T. Tate, M. Warschauer, and P. Gerjets, "Chatgpt in education: global reactions to ai innovations," *Scientific reports*, vol. 13, no. 1, p. 15310, 2023.

[255] X. Shen, Z. Chen, M. Backes, and Y. Zhang, "In chatgpt we trust? measuring and characterizing the reliability of chatgpt," 2023.

[256] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," *arXiv preprint arXiv:2212.10560*, 2022.

[257] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 21 558–21 572. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[258] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[259] "Big code leaderboard," https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard, 2024.

[260] "Gpt-4," https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4, 2023.

[261] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[262] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[263] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[264] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love *et al.*, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[265] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[266] "Models on hugging face," https://huggingface.co/models, 2024.

[267] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904. [Online]. Available: http://www.jstor.org/stable/1412159

[268] W. P. D. David M. Corey and M. J. Burke, "Averaging correlations: Expected values and bias in combined pearson rs and fisher's z transformations," *The Journal of General Psychology*, vol. 125, no. 3, pp. 245–261, 1998. [Online]. Available: https://doi.org/10.1080/00221309809595548

[269] L. Myers and M. J. Sirois, "Spearman correlation coefficients, differences between," *Encyclopedia of statistical sciences*, vol. 12, 2004.

[270] J. Hodges Jr, "The significance probability of the smirnov two-sample test," *Arkiv för matematik*, vol. 3, no. 5, pp. 469–486, 1958.

[271] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Is model attention aligned with human attention? an empirical study on large language models for code generation," *arXiv preprint arXiv:2306.01220*, 2023.

[272] C. Liu, S. D. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Codemind: A framework to challenge large language models for code reasoning," 2024. [Online]. Available: https://arxiv.org/abs/2402.09664

[273] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[274] Y. Zhuang, Q. Liu, Y. Ning, W. Huang, R. Lv, Z. Huang, G. Zhao, Z. Zhang, Q. Mao, S. Wang *et al.*, "Efficiently measuring the cognitive ability of llms: An adaptive testing perspective," *arXiv preprint arXiv:2306.10512*, 2023.

[275] M. M. Morovati, A. Nikanjam, F. Tambon, F. Khomh, and Z. M. Jiang, "Bug characterization in machine learning-based systems," *Empirical Software Engineering*, vol. 29, no. 1, p. 14, 2024.

[276] M. Taraghi, G. Dorcelus, A. Foundjem, F. Tambon, and F. Khomh, "Deep learning model reuse in the huggingface community: Challenges, benefit and trends," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 512–523.

[277] M. M. Morovati, F. Tambon, M. Taraghi, A. Nikanjam, and F. Khomh, "Common challenges of deep reinforcement learning applications development: an empirical study," *Empirical Software Engineering*, vol. 29, no. 4, p. 95, 2024.

[278] S. K. Ngassom, A. M. Dakhel, F. Tambon, and F. Khomh, "Chain of targeted verification questions to improve the reliability of code generated by llms," *arXiv preprint arXiv:2405.13932*, 2024.

[279] A. Mahu, A. Singh, F. Tambon, B. Ouellette, J.-F. Delisle, T. S. Paul, F. Khomh, A. Marois, and P. Doyon-Poulin, "Validation of vigilance decline capability in a simulated test environment: A preliminary step towards neuroadaptive control," *Neuroergonomics and Cognitive Engineering*, p. 45, 2024.

APPENDIX A CO-AUTHORSHIP

The following publications/submissions were produced during the Ph.D's degree studies:

Publications/submissions used in the thesis:

- Tambon, F., Khomh, F., & Antoniol, G. (2023). A probabilistic framework for mutation testing in deep neural networks. Information and Software Technology, 155, 107129. [19]
- Tambon, F., Majdinasab, V., Nikanjam, A., Khomh, F., & Antoniol, G. (2023, April). Mutation testing of deep reinforcement learning based on real faults. In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 188-198). IEEE. [20]
- Tambon, F., Khomh, F., & Antoniol, G. (2023). GIST: Generated Inputs Sets Transferability in Deep Learning. ACM Transactions on Software Engineering and Methodology. [21]
- Tambon F., Nikanjam A., Khomh F., and Antoniol G., "Assessing programming task difficulty for efficient evaluation of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21227 [22].

Publications/submissions mentioned in the thesis:

- Tambon, F., Laberge, G., An, L., Nikanjam, A., Mindom, P. S. N., Pequignot, Y., ... & Laviolette, F. (2022). How to certify machine learning based safety-critical systems? A systematic literature review. Automated Software Engineering, 29(2), 38. [89]
- Tambon, F., Nikanjam, A., An, L., Khomh, F., & Antoniol, G. (2024). Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. Empirical Software Engineering, 29(1), 10. [37]
- Tambon, F., Dakhel, A. M., Nikanjam, A., Khomh, F., Desmarais, M. C., & Antoniol, G. (2024). Bugs in large language models generated code. arXiv preprint arXiv:2403.08937. (Submitted, under review EMSE) [246]

Publications/submissions not mentioned in this thesis:

- Morovati, M. M., Nikanjam, A., Tambon, F., Khomh, F., & Jiang, Z. M. (2024). Bug characterization in machine learning-based systems. Empirical Software Engineering, 29(1), 14. [275]

- Taraghi M., Dorcelus G., Foundjem A., Tambon F. and Khomh F., "Deep Learning Model Reuse in the HuggingFace Community: Challenges, Benefit and Trends," 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland, 2024, pp. 512-523 [276]

- Morovati, M. M., Tambon, F., Taraghi, M., Nikanjam, A., & Khomh, F. (2024). Common challenges of deep reinforcement learning applications development: an empirical study. Empirical Software Engineering, 29(4), 95. [277]

- Ngassom, S. K., Dakhel, A. M., Tambon, F., & Khomh, F. (2024). Chain of Targeted Verification Questions to Improve the Reliability of Code Generated by LLMs. In Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware 2024). Association for Computing Machinery, New York, NY, USA, 122–130. [278]

- Mahu, A., Singh, A., Tambon, F., Ouellette, B., Delisle, J. F., Paul, T. S., ... & Doyon-Poulin, P. (2024). Validation of Vigilance Decline Capability in a Simulated Test Environment: A Preliminary Step Towards Neuroadaptive Control. Neuroergonomics and Cognitive Engineering, 45. [279]