Prioritizing Test Cases Using Box Abstraction in Deep Neural Networks

Hamzah Al Qadasi

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Prioritisation des cas de tests basé sur l'abstraction géométrique pour les réseaux de neurones profonds

Prioritizing Test Cases Using Box Abstraction in Deep Neural Networks

Présentée par :

Hamzah AL-QADASI

Direction de thèse :

Saddek BENSALEM PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
Yliès FALCONE MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse

Rapporteurs :

ANTOINE ROLLET
MAITRE DE CONFERENCES HDR, UNIVERSITE DE BORDEAUX
PANAGIOTIS KATSAROS
ASSOCIATE PROFESSOR, ARISTOTELEIO PANEPISTIMIO THESSALONIKIS

Thèse soutenue publiquement le **2 mai 2024**, devant le jury composé de :

OUM-EL-KHEIR AKTOUF, PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Présidente
SADDEK BENSALEM, PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
ANTOINE ROLLET, MAITRE DE CONFERENCES HDR, UNIVERSITE DE BORDEAUX	Rapporteur
PANAGIOTIS KATSAROS, ASSOCIATE PROFESSOR, ARISTOTELEIO PANEPISTIMIO THESSALONIKIS	Rapporteur
XIAOWEI HUANG, FULL PROFESSOR, THE UNIVERSITY OF LIVERPOOL	Examinateur

Invités :

YLIES FALCONE
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES

HAMZAH AL-QADASI

PRIORITIZING TEST CASES USING BOX ABSTRACTION IN DEEP NEURAL NETWORKS

PRIORITIZING TEST CASES USING BOX ABSTRACTION IN DEEP NEURAL NETWORKS

HAMZAH AL-QADASI

Test Prioritization Framework for Deep Neural Networks

PhD Graduate
Verimag Lab
Université Grenoble Alpes

May 2024

Hamzah AL-QADASI: *Prioritizing Test Cases Using Box Abstraction in Deep Neural Networks*,
Test Prioritization Framework for Deep Neural Networks, PhD Graduate, May 2024

ABSTRACT

In Data-Centric Artificial Intelligence (DCAI), enhancing data quality takes precedence over solely focusing on AI model improvements. This shift leads to massive unlabeled data, creating a need for effective methods to identify which data should be tested first. The thesis addresses the challenge of enhancing deep learning system robustness by focusing on testing misclassified or error-revealing data. This work presents new solutions for improving test prioritization within DCAI.

Deep learning systems, crucial in many sectors, can sometimes behave unpredictably, leading to problematic outcomes. To mitigate this problem, we introduce DeepAbstraction, a test prioritization framework designed to identify and prioritize potential error-prone instances within large, unlabeled datasets. DeepAbstraction strategically utilizes runtime monitors to create box abstractions- clusters of similar instances. This approach enhances the identification of misclassified instances, a task where traditional methods often struggle. Our research shows that DeepAbstraction performs effectively in identifying a wider range of potential error-revealing instances compared to existing methods.

Evaluating the effectiveness of test prioritization methods remains a key challenge. Existing metrics such as APFD, RAUC, and ATRC, commonly used for evaluating test prioritization techniques, have certain limitations. These metrics do not fully account for essential factors like the costs involved in labeling data. Furthermore, there is an excessive focus on the rate at which faults are detected, overshadowing the equally important aspect of the ratio of detected faults. This oversight suggests a gap in how these metrics reflect the true performance of test prioritization algorithms. Our research addresses these limitations by introducing two novel metrics: the Weighted Fault Detection Ratio (WFDR) and the Severity Fault Detection Rate (SFDR). WFDR improves assessment by balancing the fault detection ratio and rate. Furthermore, SFDR focuses on prioritizing instances with high-severity misclassifications. Collectively, these metrics provide a more thorough evaluation of test prioritization techniques.

The final part of this thesis revisits the limitations of the initial DeepAbstraction framework. A key challenge in the DeepAbstraction framework lies in selecting the right τ value—a parameter that affects the size of *boxes* used to categorize data. The selection of τ is crucial as it directly influences the framework's stability and effectiveness. To address this issue, we develop a method that combines all multiple monitors into one comprehensive monitor to assess network predictions. This ensures that no single assessment entirely controls the decision-making process, unlike the previous version of DeepAbstraction. We also introduce several techniques to integrate the verdicts of monitors into a final, balanced verdict. Our approach significantly improves the performance and stability of the DeepAbstraction framework. In comparison with leading algorithms, our enhanced version, DeepAbstraction++, consistently marks an improvement in performance by 2.38% to 7.71%.

In conclusion, this thesis makes a significant contribution to the field of test prioritization in Data-Centric AI. It introduces a robust framework and two metrics while also addressing existing limitations.

RÉSUMÉ

Dans l'intelligence artificielle centrée sur les données (Data-Centric AI, DCAI), l'amélioration de la qualité des données prend le pas sur l'amélioration des modèles d'IA. Ce changement conduit à des données non étiquetées massives, créant un besoin de méthodes efficaces pour identifier les données qui devraient être testées en premier. La thèse aborde le défi de l'amélioration de la robustesse des systèmes d'apprentissage profond en se concentrant sur le test des données mal classées ou révélatrices d'erreurs. Ce travail présente de nouvelles solutions pour améliorer la priorisation des tests au sein de DCAI.

Les systèmes d'apprentissage profond, essentiels dans de nombreux secteurs, peuvent parfois se comporter de manière imprévisible, ce qui entraîne des résultats problématiques. Pour atténuer ce problème, nous présentons DeepAbstraction, un cadre de hiérarchisation des tests conçu pour identifier et hiérarchiser les instances potentiellement sujettes à des erreurs dans de grands ensembles de données non étiquetées. DeepAbstraction utilise stratégiquement des moniteurs d'exécution pour créer des abstractions de boîtes - des grappes d'instances similaires. Cette approche améliore l'identification des instances mal classées, une tâche pour laquelle les méthodes traditionnelles ont souvent des difficultés. Nos recherches montrent que DeepAbstraction permet d'identifier efficacement un plus grand nombre d'instances potentiellement révélatrices d'erreurs que les méthodes existantes.

L'évaluation de l'efficacité des méthodes de hiérarchisation des tests reste un défi majeur. Les métriques existantes telles que APFD, RAUC et ATRC, couramment utilisées pour évaluer les techniques de hiérarchisation des tests, présentent certaines limites. Ces métriques ne tiennent pas pleinement compte de facteurs essentiels tels que les coûts liés à l'étiquetage des données. En outre, l'accent est mis de manière excessive sur le taux de détection des fautes, occultant l'aspect tout aussi important du ratio des fautes détectées. Cette omission suggère une lacune dans la manière dont ces métriques reflètent la véritable performance des algorithmes de hiérarchisation des tests. Notre recherche aborde ces limitations en introduisant deux nouvelles métriques : le ratio pondéré de détection des fautes (Weighted Fault Detection Ratio, WFDR) et le taux de détection des fautes de gravité (Severity Fault Detection Rate, SFDR). Le WFDR améliore l'évaluation en équilibrant le ratio et le taux de détection des fautes. En outre, le SFDR se concentre sur la priorisation des instances présentant des erreurs de classification de grande gravité. Collectivement, ces métriques fournissent une évaluation plus approfondie des techniques de hiérarchisation des tests.

La dernière partie de cette thèse revient sur les limites du cadre initial de DeepAbstraction. L'un des principaux défis du cadre de DeepAbstraction réside dans la sélection de la bonne valeur de τ - un paramètre qui affecte la taille des *boxes* utilisées pour catégoriser les données. Le choix de τ est crucial car il influence directement la stabilité et l'efficacité du cadre. Pour résoudre ce problème, nous développons une méthode qui combine tous les moniteurs multiples en un seul moniteur complet pour évaluer les prédictions du réseau. Cela garantit qu'aucune évaluation ne contrôle entièrement le processus de prise de décision, contrairement à la version précédente de DeepAbstraction. Nous introduisons également plusieurs techniques pour intégrer les verdicts des moniteurs dans un verdict

final équilibré. Notre approche améliore considérablement les performances et la stabilité du cadre de DeepAbstraction. En comparaison avec les principaux algorithmes, notre version améliorée, DeepAbstraction++, marque une amélioration constante des performances de 2,38 % à 7,71 %.

En conclusion, cette thèse apporte une contribution significative au domaine de la hiérarchisation des tests dans l'IA centrée sur les données. Elle introduit un cadre robuste et deux métriques tout en abordant les limitations existantes.

PUBLICATIONS

This thesis incorporates several ideas and figures that have been previously published in the following works:

[1] Hamzah Al-Qadasi, Changshun Wu, Yliès Falcone, and Saddek Bensalem. "DeepAbstraction: 2-Level Prioritization for Unlabeled Test Inputs in Deep Neural Networks." In: *IEEE International Conference On Artificial Intelligence Testing, AITest 2022, Newark, CA, USA, August 15-18, 2022*. IEEE, 2022, pp. 64–71.

[2] Hamzah Al-Qadasi, Yliès Falcone, and Saddek Bensalem. "DeepAbstraction++: Enhancing Test Prioritization Performance via Combined Parameterized Boxes." In: *2023 International Symposium On Leveraging Applications of Formal Methods (AISoLA)*. Springer. 2023.

[3] Hamzah Al-Qadasi, Yliès Falcone, and Saddek Bensalem. "Difficulty and Severity-Oriented Metrics for Test Prioritization in Deep Learning Systems." In: *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE. 2023, pp. 40–48.

ACKNOWLEDGMENTS

First and foremost, I express my profound gratitude to Prof. Saddek Bensalem, my supervisor, for his unwavering support, invaluable guidance, and constant encouragement throughout this challenging yet rewarding journey. His wisdom and mentorship have been pivotal in shaping my academic and research pursuits.

I am equally thankful to Prof. Yliès Falcone, my co-supervisor, whose insights, expertise, and feedback have been crucial in refining my work and enhancing my research skills. His dedication and commitment to excellence have inspired me to strive for nothing but the best.

My sincere appreciation goes to Prof. Antoine Rollet and Prof. Panagiotis Katsaros, my Ph.D. thesis reporters, for their rigorous review and constructive critiques of my thesis. Their feedback has significantly contributed to the improvement and depth of my research. I extend my gratitude to Prof. Oum-El-Kheir Aktouf and Prof. Xiaowei Huang, my PhD defense examiners, for their thoughtful examination and valuable suggestions, which have further enriched my academic endeavors.

I cannot express enough thanks to my family: my father, Abduljalil, my mother, Huda, my wife, Safa, and my brothers and sisters for their endless love, understanding, and support. They have been my rock, my source of strength and motivation, throughout this journey and beyond.

Last but certainly not least, I thank my best friends: Taha, Ayman, Bilal, and Yousif, for their camaraderie, encouragement, and unwavering support. They have been a constant source of joy and comfort, making this journey all the more memorable.

To all of you, I extend my deepest gratitude. This achievement is not solely mine but a testament to the collective effort, belief, and support of each one of you. Thank you for being a part of my journey.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

DCAI Data-Centric AI

DNNs Deep Neural Networks

MT Metamorphic Testing

MCAI Model-Centric AI

DL Deep Learning

MRs Metamorphic Relations

GAN Generative Adversarial Network

BRC Background-Relevance

DL Deep Learning

ML Machine Learning

WFDR Weighted Fault Detection Ratio

SFDR Severity Fault Detection Rate

DNN Deep Neural Network

CNN Convolutional Neural Network

CNNs Convolutional Neural Networks

ReLU Rectified Linear Unit

FCL Fully Connected Layer

APFD Average Percentage of Faults Detection

RAUC Ratio Area Under Curve

ATRC Average Test Relative Coverage

OOD Out-Of-Distribution

GI Gini Impurity or Gini Index

TRC Test Relative Coverage

FDRE Fault Detection RatE

FDRO Fault Detection RatiO

DSA Distance-based Surprise Adequacy

SOTA State-Of-The-Art

MR Misclassification Ratio

FD+ Familiarity score

BNNs Bayesian Neural Networks

DSA Distance-based Surprise Adequacy

MLP Multi-Layer Perceptron

XAI Explainable AI

SHAP SHapley Additive exPlanations

LIME Local Interpretable Model-Agnostic Explanations

Grad-CAM Gradient-weighted Class Activation Mapping

DEFINITIONS AND TERMINOLOGY

To ensure clarity and coherence, this section briefly defines key terms and concepts used throughout the thesis:

- **Data-Centric AI:** An approach that emphasizes the importance of high-quality data over model architecture for improving AI system performance.
- **Big Data:** Extremely massive datasets that may be analyzed to reveal patterns, trends, and associations, which challenge traditional data processing approaches.
- **High-Stakes Domains:** Areas such as healthcare and autonomous driving where errors or vulnerabilities in AI systems can result in severe consequences, including loss of life.
- **Data Annotation:** The process of attaching labels or metadata to unstructured data, such as images or text, to make it usable for machine learning models.
- **Test Prioritization:** A technique aimed at ranking test instances based on their likelihood to reveal system vulnerabilities, thus optimizing resource allocation during the data labeling and validation process.
- **Model Reliability:** Is the ability of a machine learning model to maintain consistent performance levels and reliably produce correct outputs. This characteristic signifies the model's dependability across different applications and its stable performance over time.
- **Model Robustness:** The ability of a machine learning model to perform well under various conditions, including the presence of noise or previously unseen data.
- **Performance Stability (Hyperparameters)**: A machine learning model's ability to maintain consistent accuracy or other metrics across different hyperparameter configurations. High-performance stability indicates less sensitivity to hyperparameter variations.
- **ML Model Deployment:** The process of integrating a trained machine learning model into a production environment where it can process real-world data and provide actionable insights or decisions.
- **Data Cascades:** Compounding issues in data-driven projects that arise from initial errors in how data is collected, stored, or used, often leading to suboptimal results in AI systems.
- **Data Management Systems:** The frameworks and technologies used for storing, retrieving, and managing data, which are crucial for the efficient operation of AI and machine learning systems.

Part I

INTRODUCTION AND BACKGROUND

1 INTRODUCTION

1.1 MOTIVATION

The emerging idea of Data-Centric AI (DCAI) has greatly elevated the importance of data in AI systems. This approach stands in contrast to Model-Centric AI (MCAI), which primarily concentrates on developing and fine-tuning Deep Neural Networks (DNNs). DCAI shifts the focus from merely fine-tuning models to systematically engineering the data that feeds the models, especially in high-stakes domains like healthcare and autonomous driving. This shift comes from recognizing that even the most advanced algorithms can yield suboptimal results and trigger data-cascades [67] if the training data is flawed, biased, or incomplete. To address this, DCAI aims to ensure that data is accurate, well-labeled, and reflective of real-world scenarios. For instance, in healthcare, better data can lead to more accurate diagnoses[9], while in autonomous driving, high-quality data can significantly improve safety measures. By focusing on data quality, DCAI offers a practical, scalable, and cost-effective path to improve reliability, fairness, and overall performance across a wide array of applications.

With the advent of big data, the landscape for innovative research and applications has expanded dramatically. For instance, sectors ranging from healthcare to finance are generating petabytes of data at an unprecedented rate. This massive volume of the collected data can significantly enhance intelligent systems, facilitating more precise predictions and improved decision-making. However, the manual annotation of large datasets proves to be a labor-intensive, costly, and time-consuming task. These challenges become even more pronounced when specialized domain knowledge is necessary for accurate data labeling, e.g., healthcare [52] and aviation [18]. Additionally, the complexity increases when a single test instance includes multiple objects for labeling, a common scenario in object detection and segmentation problems. Advanced AI systems seem capable of managing large data volumes effortlessly. However, the actual limitation often exists in the foundational data management systems responsible for storage and retrieval, as well as in the availability and capacity of processing units crucial for efficiently running these systems. Inadequacies in these systems can lead to operational bottlenecks and errors. Such issues undermine the efficiency of AI systems that depend on these data management systems. Therefore, the development of robust labeling strategies becomes essential, consequently enhancing the potential of big data and improving the performance of DCAI systems.

In situations with enormous data volumes and restricted labeling budgets, the random selection of test instances for labeling is problematic. This approach often leads to the neglect of specific instances crucial for revealing vulnerabilities in Deep Learning (DL) systems. If these critical but neglected test instances are not included during the testing phase, the system might lack robustness and fail in challenging or novel real-world scenarios. For instance, the fatal incident where a Tesla vehicle failed to identify a trailer due to a rare combination of lighting and height conditions [78]. Similarly, a Google self-driving car was involved in a collision with a bus, triggered by an unpredictable sequence of events [79]. Therefore, this oversight carries significant implications, including catastrophic failures and, in extreme situations, loss of life. As a result, this situation

underscores the critical role of strategic test prioritization. As a result, this can significantly reduce the risks associated with deploying intelligent systems in real-world scenarios.

Test prioritization is a systematic technique designed to rank unlabeled test instances in order of their likelihood to reveal errors, vulnerabilities, or other points of interest in a given system. Moreover, test prioritization aims to optimize the process of data labeling and validation by identifying the most critical and relevant data points for review. This approach not only saves time but also allocates resources more efficiently like computational power and human expertise. In high-stakes domains, where the margin for error is minimal, test prioritization becomes indispensable. It ensures that the most critical data, which exposes system vulnerabilities, receives immediate attention, thereby substantially elevating the system's reliability and robustness.

For example, in healthcare, test prioritization could help identify the most ambiguous or complex medical images that are likely to challenge an AI diagnostic system. By giving priority to these challenging cases for expert human review, the system can better learn to handle such complexity. This strategy thereby improves its diagnostic accuracy and robustness over time. Similarly, in autonomous driving, test prioritization can focus on complex or high-risk driving scenarios that an AI system struggles to handle. Recognizing these challenging cases early allows developers to direct additional resources toward labeling these particular instances. This focused effort improves the AI system's competency in handling difficult conditions, which in turn elevates road safety. The adoption of test prioritization strategies enables organizations to fine-tune their system's performance while optimizing the use of available resources.

1.2 SCOPE

The scope of this thesis centers on improving test prioritization's effectiveness and stability in DCAI systems. Test prioritization is crucial in scenarios where data quality and timely labeling are paramount. The research introduces a practical framework, *DeepAbstraction++*, and two novel evaluation metrics, to enhance the data labeling and validation process. Moreover, it highlights the significance of test prioritization algorithms in debugging AI systems before deployment, with a particular focus on multi-class classification problems and image-based applications.

1.3 TEST GENERATION TECHNIQUES

Test generation techniques for DNNs are critical methodologies that ensure the performance, reliability, and robustness of these advanced systems. These techniques, each addressing unique aspects of DNN testing, become even more effective when combined with test prioritization strategies. This combination enhances the overall testing process. Key techniques in this realm include Metamorphic Testing, Mutation Testing, and Fuzz Testing, each offering distinct advantages and approaches to DNN testing. These test generation techniques, augmented with test prioritization, provide a robust framework for DNN assessment, ensuring early issue detection, optimal resource allocation, and the development of reliable DNN models. Each technique addresses different testing challenges, contributing uniquely to the comprehensive validation of DNNs.

1.3.1 *Metamorphic Testing*

Metamorphic Testing (MT) [62, 93, 94] in the context of DL-driven software, particularly those processing images, is an innovative approach adapted from traditional software engineering. The essence of this method lies in defining Metamorphic Relations (MRs) and using these MRs to generate test images. MRs serve as guidelines that dictate how changes in input (such as images) should affect the system's output. In the following, we explore key examples of Metamorphic Testing:

- **DeepTest:** [77] This method is used in DNN-driven autonomous driving systems. It generates test input images through transformations like changing brightness, and contrast, or adding effects such as raindrops. However, the realism of these transformed images is limited.
- **DeepRoad:** [89] DeepRoad uses a Generative Adversarial Network (GAN)-based image transformation method to create more realistic images of diverse driving scenes, e.g., rainy or snowy roads. These images are then used to test the consistency of the autonomous driving system's performance.
- **DeepBackground:** [14] Focusing on the influence of image backgrounds, DeepBackground introduces the Background-Relevance (BRC) metric. It generates test images by altering the background while keeping the main object unchanged, assessing the robustness of image recognition systems against background variations.

These techniques illustrate how metamorphic testing can be applied to test various aspects of DL systems, emphasizing general performance under different conditions and robustness against background changes. Given that metamorphic testing can generate an extensive dataset, the implementation of test prioritization becomes crucial. This involves strategically organizing test cases in order of importance, focusing on the criticality and complexity of the metamorphic relations. Such prioritization ensures that the most significant scenarios are examined first, thereby optimizing the efficiency and effectiveness of the testing process.

1.3.2 *Mutation Testing*

Mutation testing [47] in the context of Machine Learning (ML) and DL involves the systematic introduction of small modifications or *mutations* to the components of these systems. This method aims to evaluate the efficacy of test suites in identifying these induced errors. In conventional software engineering, mutation testing typically entails making incremental alterations to the program's source code. However, within the realm of DL systems, this process extends beyond mere code modification. It encompasses variations in the testing dataset and also alterations to the neural network architectures. There are many leading research works in this area, e.g., DeepMutation [42], and DeepMutation++ [29], DeepCrime [30]

For example, mutation testing includes manipulating the characteristics of the input data, injecting noise, or modifying the layers and neurons in the neural network. These deliberate perturbations serve as proxies for potential real-world errors or anomalies. The principal objective of this approach is to assess the resilience and accuracy of deep learning

systems. A test suite that effectively identifies and addresses these simulated mutations is deemed capable of reliably detecting and handling actual, unforeseen faults. Thus, mutation testing in deep learning is a strategic approach to ensure the robustness and dependability of these advanced computational models.

Test prioritization with mutation testing creates a synergistic approach to enhancing the efficiency and effectiveness of AI testing. This integration involves organizing test cases in a manner that prioritizes the detection of critical faults. This prioritization is crucial when dealing with a large number of mutations introduced during mutation testing. It aids in identifying those test cases that are particularly effective at detecting significant mutations, ensuring early detection of vital issues.

1.3.3 *Fuzz Testing*

Fuzz testing involves systematically a wide variety of modified inputs into neural network models to test their robustness and error-handling capabilities. In practice, fuzz testing creates inputs that are diverse and often slightly altered. For example, in a neural network tasked with image recognition, fuzz testing would involve inputting images with various modifications, e.g., adding random noise, applying filters that distort or obscure parts of the image, or altering color schemes in ways that are not typically encountered during the network's training phase.

The value of fuzz testing in deep learning is highlighted by its ability to reveal how these models react to unexpected or *noisy* data, a common occurrence in real-world scenarios. By determining the points of failure or performance degradation in these models under such conditions, developers and researchers can gain insights into potential weaknesses or blind spots. This leads to more robust and error-resilient neural network models. These networks are particularly important in applications where reliability and accuracy are critical, such as in medical diagnostics, and autonomous vehicles. The leading works in this area are DeepHunter [86], FuzzGAN [26], DLRegion [76], and GradFuzz [51]. Test prioritization in fuzz testing involves identifying and testing scenarios most likely to reveal significant vulnerabilities first. This strategy ensures that the most critical security flaws are detected early, enabling timely remediation and reinforcing the overall security posture of the DNN systems.

1.4 SUMMARY OF CONTRIBUTIONS

The thesis has three main contributions to the field of test prioritization. After thorough studying for the state-of-the-art work, there has been very little research work published in this area until 2020. Our first contribution was to enrich this research area with an effective framework namely, DeepAbstraction. DeepAbstraction focuses on improving test prioritization area for deep learning systems. This contribution includes the following:

1. **Effectiveness of monitors in test prioritization**: The paper is the first to investigate the role of runtime monitors in the area of test prioritization for deep learning systems.

2. **Comprehensive study on misclassified instances**: The paper provides an in-depth study of where misclassified instances can reside in the feature space. It discusses

the limitations of existing techniques that focus mainly on near-boundary instances and introduces a way to also prioritize near-centroid instances effectively.

3. **State-of-the-art performance**: DeepAbstraction is empirically shown to outperform existing state-of-the-art test prioritization techniques. It does not require any prior labeling for test instances to operate, making it a practical solution for reducing labeling costs.

After publishing the first paper, we identified some limitations in the existing metrics like APFD, RAUC, and ATRC, which are commonly used to evaluate test prioritization algorithms. Thus, we focused on developing new metrics to evaluate effectively the test prioritization techniques in deep learning systems. More specifically, we proposed two new metrics: Weighted Fault Detection Ratio (WFDR) and Severity Fault Detection Rate (SFDR). The main contributions of the second paper are as follows:

1. **Intensive study on existing metrics:**
   The paper conducts the first intensive study to investigate the effectiveness of existing metrics for test prioritization in deep learning systems. It highlights the limitations of these metrics, such as neglecting the labeling budget and prioritization difficulty.

2. **Introduction of WFDR metric:**
   The paper introduces a new metric called WFDR that addresses the limitations of existing metrics. WFDR evaluates algorithms based on both fault detection ratio and rate. It also incorporates the prioritization difficulty adaptively during the prioritization process.

3. **Introduction of SFDR metric:**
   The paper also introduces another metric called SFDR, which evaluates algorithms in the context of severity prioritization. This metric is particularly useful in critical situations where prioritizing misclassified instances according to their severity level is crucial.

In the third paper, we revisited the first paper to address the limitations of the DeepAbstraction framework. We introduced *combined parameterized boxes* to improve the selection issue of the τ parameter, enhancing the framework's performance and stability. The main contributions of the paper are as follows:

1. **Identification of weaknesses in the earlier version of DeepAbstraction:** The paper starts by analyzing the earlier version of DeepAbstraction, pointing out the weaknesses that compromise its performance and reliability. Specifically, it highlights the challenge of selecting the appropriate τ parameter, which affects the size of the boxes used for abstraction.

2. **Introduction of combined parameterized boxes:** We also introduce a new methodology called *combined parameterized boxes*. This approach uses multiple monitors with various τ values to evaluate network predictions. It aims to overcome the limitations of relying on a single monitor's verdict, thereby enhancing the accuracy of the system's decisions.

3. **Unique weighting system and combination strategy:** The paper establishes a unique weighting system that balances the decision-making process when conflicts arise

among different verdicts of monitors. This system assigns unique weights to rejection, acceptance, and uncertainty verdicts. We also propose multiple strategies for integrating these weighted verdicts into a conclusive verdict, such as mean, max, product, and mode.

1.5 THESIS OUTLINE

This document is organized into four parts.

1.5.1 *Part I: Introduction and Background*

- **Chapter 1**: Introduces the thesis topic and provides context and motivation. Also includes a summary of contributions.
- **Chapter 2**: Offers background information on key topics like deep neural networks, test prioritization, and runtime monitoring.

1.5.2 *Part II: Contributions*

- **Chapter 3**: Introduces *DeepAbstraction*, a 2-level prioritization algorithm for unlabeled test inputs. Discusses its effectiveness, efficiency, and stability.
- **Chapter 4**: Focuses on metrics for test prioritization in deep learning, specifically targeting the faults severity and prioritization difficulty.
- **Chapter 5**: Enhances the initial framework through *DeepAbstraction*++, which combines parameterized boxes for better performance.

1.5.3 *Part III: Related Work*

- **Chapter 6**: Reviews the current state of the art and related work in test prioritization algorithms and introduces concepts in runtime monitoring.

1.5.4 *Part IV: Conclusion and Future Work*

- **Chapter 7** outlines the final conclusion and discusses potential avenues for future research.

2

BACKGROUND

In this chapter, the focus is on laying the foundational concepts that guide the entire research. It begins with an exploration of features and feature space, delving into the intricacies of DNNs and Convolutional Neural Networks (CNNs). The chapter also covers essential topics such as multi-class classification, test prioritization, and runtime monitoring. We discuss various statistical scoring functions, including Gini Impurity and Entropy providing an in-depth insight into the topic. Furthermore, the chapter extends the scope to include evaluation metrics like APFD, RAUC, and ATRC. It concludes with areas closely related to test prioritization, such as active learning and test selection. This chapter functions not only as a foundational context but as a roadmap, connecting diverse concepts and laying the groundwork for the in-depth exploration that follows.

2.1 FEATURES

In deep learning, features [90] are often represented as feature vectors, which are n-dimensional vectors of numerical features that represent an object. These feature vectors capture the relevant information about the object and serve as input to the classifier. The feature space is the vector space associated with these feature vectors.

For instance, consider a gray-scale image, sized 28x28 pixels. In deep learning, this image is represented as a 784-dimensional feature vector, where each dimension corresponds to the intensity of a pixel in the image. The pixel intensities, ranging from 0 to 255, form the numerical values of the feature vector. This vector serves as the input to a classifier, encapsulating all necessary information about the image within the 784-dimensional feature space.

2.2 FEATURE SPACE

The feature space [80] in deep learning refers to the space or representation where the input data is transformed and processed by the deep learning model. It is a high-dimensional space that captures the important features or patterns in the data. Each dimension in the feature space corresponds to a specific feature or attribute of the input data.

Deep learning models learn to extract meaningful features from the raw input data through a series of layers, such as convolutional layers, recurrent layers, or fully connected layers. These layers transform the input data into a more abstract and compact representation in the feature space. The goal is to create a representation that is more suitable for the task at hand, such as image classification or natural language processing.

For example, in a facial recognition deep learning model, the input is colored face images. These images first appear as pixel-level features. As they pass through convolutional layers of the model, these simple features transform into complex ones like edges, textures, and facial components (eyes, nose). In this context, the feature space is a high-dimensional space where each dimension represents a specific, learned facial feature. This transformation allows the model to effectively differentiate and recognize individual faces based on characteristics like eye shape or smile, identified in distinct dimensions of the feature space.

The feature space plays a crucial role in deep learning as it determines the quality and effectiveness of the learned representations. A well-designed feature space can lead to better performance and generalization of the deep learning model. Researchers often explore different techniques and architectures to improve the data representation in the feature space and enhance the performance of deep learning models.

In deep learning, the feature space effectively captures and transforms input data. Building on this concept, exploring the feature space is critical not just for enhancing data representation in models but also for identifying potential inadequacies. This exploration directly relates to how features are represented and transformed within the model. By examining each dimension of this complex space, where each dimension represents a unique aspect of the input data, researchers gain insights. They understand how specific input features influence the model's decision-making process. Such thorough analysis of the feature space is essential for uncovering weaknesses or limitations in the model's decision-making process, crucial for developing more robust and reliable deep learning models [96].

2.3 DEEP NEURAL NETWORK

A Deep Neural Network (DNN) is a type of artificial neural network with multiple layers between the input and output layers as shown in Fig. 2.1. These intermediate layers, known as hidden layers, allow the network to learn complex patterns and representations of the input data. Let X_t and X_s denote the training and test datasets, respectively, where $X_t = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$, where m is the number of the training instances, x is a network input, and y is the ground truth or the actual class. Similarly, let $X_s = \{x_1, x_2, ..., x_r\}$, where x is an unlabeled test input and r is the number of test instances. Let $\mathcal{N}$ be a neural network, and $\mathcal{N}(x)$ be the network prediction. $\mathcal{N}$ consists of a set of layers such that $L = \{L_k \mid 1 \leq k \leq q\}$, where q is the total number of layers in the neural network. The number of neurons in a layer L_k is denoted by $|L_k|$. In a multi-class classification problem, the last layer L_q is a softmax layer, and L_{q-1} layer, namely the penultimate layer.

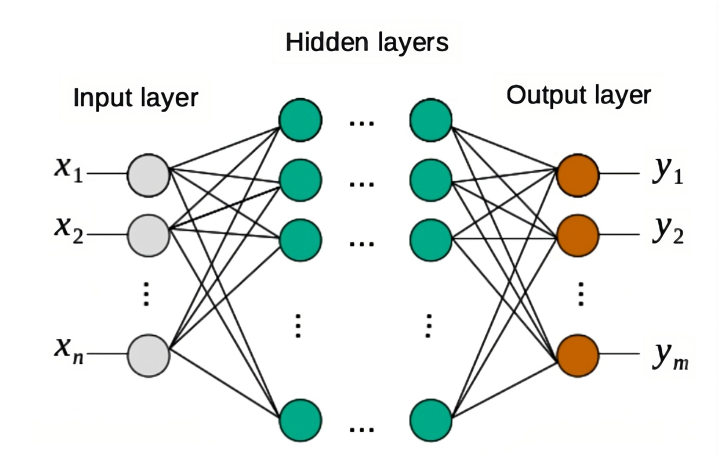


Figure 2.1: An example of FeedForward Neural Network.

EXAMPLE 1: *Consider a deep neural network designed for image classification. The network denoted as $\mathcal{N}$, consists of multiple layers, including input, hidden, and output layers. $\mathcal{N}$ has a total of q layers, with the last layer L_q being a softmax layer for classification.*

The training dataset, X_t, includes pairs of images and their corresponding labels, (x_i, y_i), where x_i represents an image, and y_i is its label. For instance, if the task is to classify animals, x_i could be an image of a cat, and y_i would be the label "cat". There are m such pairs in X_t.

The test dataset, X_s, comprises unlabeled images $\{x_1, x_2, ..., x_r\}$ used to evaluate the model. Each layer L_k in $\mathcal{N}$ transforms the input data, with the number of neurons in each layer denoted as $|L_k|$. Ultimately,$\mathcal{N}$ categorizes the input image into classes like "cat", "dog", etc., in the softmax layer on the learned patterns.

2.4 CONVOLUTIONAL NEURAL NETWORKS

CNNs [37] are specialized neural networks designed for processing data with a grid-like topology, such as an image. They excel in various computer vision tasks including image classification [61], object detection [63], image segmentation [66], medical image analysis [16], and autonomous driving [3].

2.4.1 *Convolutional Neural Network*

A Convolutional Neural Network (CNN) is composed of several interconnected layers, each tailored for specific tasks within the realms of image processing and analysis. The foundational element of a CNN is its convolutional layers. These layers are responsible for performing the convolution operation, a pivotal process for extracting features from images. The convolution operation can be mathematically expressed as:

$$G(i,j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I(i-u, j-v) \cdot K(u,v)$$

In this formula, I denotes the input image, represented as a matrix or an array. Each element, $I(x,y)$, within this matrix corresponds to the pixel value at the specific position (x,y) in the image. This granular approach to the image allows the convolution process to methodically analyze it by assessing the intensity or color information at each pixel. The kernel, symbolized by K, is a smaller matrix used to extract features from the image. It interacts with the input image by overlaying itself at various positions and computing the sum of the element-wise products between the kernel's values and the corresponding pixel values of the image.

The dimensions of the kernel are represented by m and n, where m denotes the number of rows and n the number of columns in the kernel. This means that for every position (i,j) on the output feature map G, the convolution operation involves a summation process over a $m \times n$ window of the input image. Each element $I(i-u, j-v)$ indicates the pixel value at position $(i-u, j-v)$ of the input image, which interacts with the corresponding element $K(u,v)$ in the kernel. The result of this convolution is the feature map G, which emphasizes specific features or patterns present in the input image as detected by the kernel. Figure 2.2 provides a visual representation of this process, showing the systematic application of the kernel across the entire input image to generate the feature map.

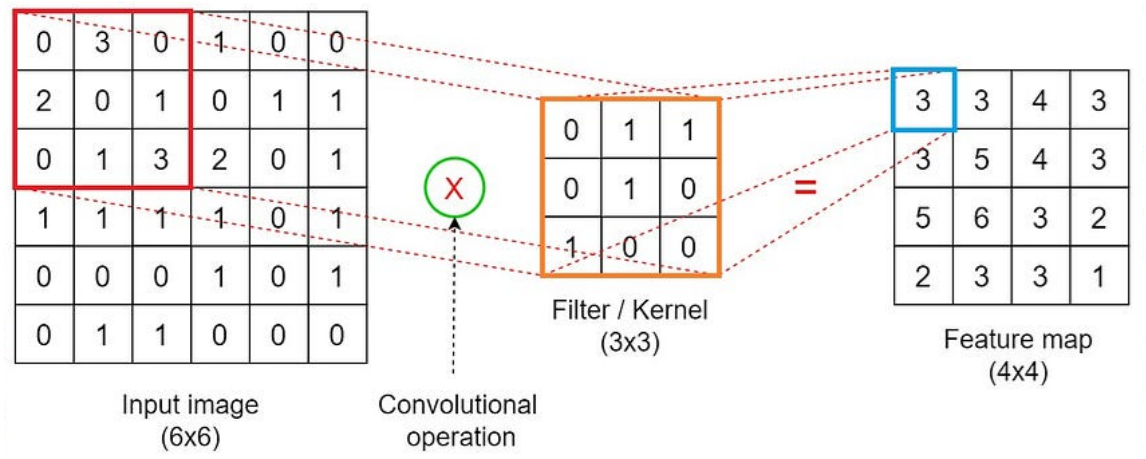


Figure 2.2: An example of convolution operation with a kernel size of m×n=3×3.

2.4.2 *Activation Functions*

Following the convolutional layers, activation functions such as the Rectified Linear Unit (ReLU) introduce non-linear properties to the system. The ReLU function is defined as $f(x) = \max(0, x)$, which means it retains positive values as they are while converting any negative value to zero as illustrated in Fig 2.3.

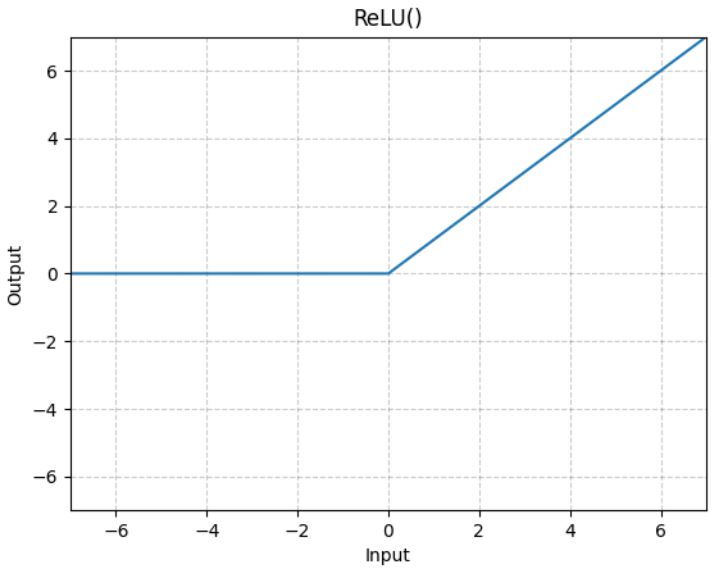


Figure 2.3: ReLU activation function.

In CNNs, when determining the activation for each element of the feature map G at a position (i, j), the ReLU activation function plays a crucial role. This process, based on the outcome of the convolution operation $G(i, j)$, is mathematically expressed as:

$$A(i, j) = \max(0, G(i, j))$$

This ReLU function is applied to each element of the feature map. In practical terms, this means that any negative values in the feature map G are set to zero. This action effectively eliminates any negative outcomes from the convolution process, ensuring they do not adversely affect the network's learning. On the other hand, the positive values in G — those that represent the detected features in the input image — are retained as they are.

Thus, the application of the ReLU function serves two pivotal purposes. First, it preserves the essential elements of the feature map that are crucial for the representation of detected features in the image. Second, it introduces a necessary non-linearity to the neural network's learning process. This non-linear characteristic is vital for neural networks to learn complex data patterns effectively. Without such non-linear transformations, the network, regardless of its depth, would be limited to functioning as a linear model, which is insufficient for complex problem-solving. The impact of the ReLU function, as well as other activation functions, on the performance of neural networks is significant. For a deeper understanding of these effects, the reader could refer to a detailed review and comparison of various activation functions as presented in [20].

2.4.3 *Pooling Layer*

Next, pooling layers [4] in CNNs are used to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer. They work by summarizing the presence of features in patches of the feature map. A common pooling operation is max pooling, which takes the maximum value of the pixels in the window being considered.

$$P(A) = \max_{i,j \in \text{window}} A(i,j)$$

In this equation, the term *window* refers to the subset of the image or feature map over which the maximum value is computed. As depicted in Fig. 2.4, max pooling is applied to each distinct 2×2 area of the input feature map, systematically reducing its size. This process highlights the most significant elements of each region, allowing the network to focus on the most prominent features while maintaining computational efficiency and robustness to variations in the input data.

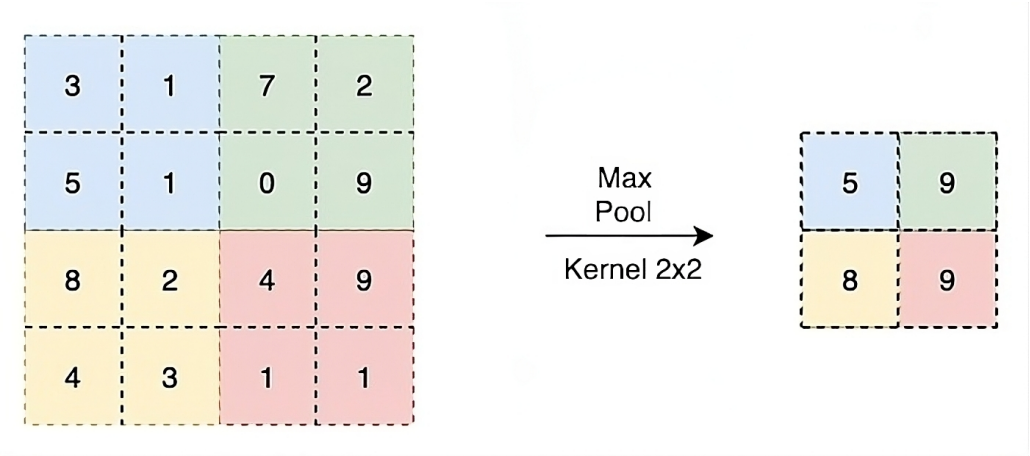


Figure 2.4: An example of max pool operation with a kernel size of 2×2.

2.4.4 *Fully Connected Layer & Output Layer*

Fully Connected Layer (FCL) connects every neuron in one layer to every neuron in the next layer, forming a dense network of connections. This structure allows the network to integrate the learned features from previous layers and make complex inferences about the input data.

Finally, the output layer produces the final prediction or classification. This layer is responsible for generating the final prediction or classification that represents the network's interpretation of the input data. Often, the softmax function is employed in this layer to facilitate the task of classification, particularly in tasks involving multiple classes.

The softmax function is formally defined as:

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$$

for $i = 1, 2, ..., K$ and an output vector y of length K.

In this equation:

- y_i is the i-th element of the output vector to the softmax function.
- K is the total number of classes.
- e^{y_i} is the exponential function applied to y_i.
- The denominator $\sum_{j=1}^{K} e^{y_j}$ is the sum of the exponential values of all elements in the output vector.

The role of softmax in the output layer is crucial for classification tasks. It converts the raw output scores, often referred to as *logits*, into probabilities by normalizing them in a way that their sum equals one. Each element of the output vector of softmax represents the probability that the input belongs to one of the K classes. The class with the highest probability is typically taken as the model's prediction.

2.4.5 *CNN layers*

These components work in unison to transform the input through a series of mathematical operations, leading to the final prediction or classification. The depth of the network, or the number of hidden layers, allows for more complex modeling of the input data. However, greater depth can make the network more challenging to train and more prone to overfitting if not properly regularized.

2.5 MULTI-CLASS CLASSIFICATION

Multi-class classification is a type of classification task where the goal is to categorize the given input into one of three or more classes. Unlike binary classification, where there are only two possible outcomes.

Given a set of features $X = \{x_1, x_2, \ldots, x_n\}$ and a set of classes $C = \{c_1, c_2, \ldots, c_k\}$, where $k > 2$, the goal of multi-class classification is to find a function $f : X \to C$ that maps each feature vector x_i to a class c_j in C.

The function f can be represented by a model trained on a set of labeled examples $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each y_i is a class label in C.

In mathematical terms, the objective is often to minimize a loss function L over the training data:

$$\min \sum_{i=1}^{n} L(f(x_i), y_i)$$

Here, L is a loss function that quantifies the difference between the model's predictions and the true class labels. A common example of a loss function used in multi-class classification is the cross-entropy loss. This loss function is defined as:

$$L(y, \hat{y}) = -\sum_{j=1}^{k} y_j \log(\hat{y}_j)$$

In this formula, y represents the true label in a one-hot encoded vector, and $\hat{y}$ represents the predicted probabilities for each class. The cross-entropy loss effectively measures the disparity between the predicted probability distribution and the true distribution. It also penalizes predictions that diverge significantly from the actual label.

EXAMPLE 2: *consider a multi-class classification problem with three classes (k=3). Suppose for a particular instance, the true class label y is the second class. In a one-hot encoded format, this label is represented as* $y = [0, 1, 0]$. *Assume the model predicts the probabilities for each class for this instance as* $\hat{y} = [0.2, 0.7, 0.1]$. *The cross-entropy loss for this prediction can be calculated as:*

$$L(y, \hat{y}) = -\sum_{j=1}^{3} y_j \log(\hat{y}_j) = -[0 \cdot \log(0.2) + 1 \cdot \log(0.7) + 0 \cdot \log(0.1)] = -\log(0.7) \approx 0.357$$

In this example, the loss is approximately 0.357, reflecting the degree of disparity between the model's prediction and the actual label. The model's objective during training is to minimize this loss, which, in this case, would involve adjusting its parameters to increase the predicted probability of the second class and decrease the probabilities of the other classes for similar instances.

2.6 TEST PRIORITIZATION

In the realm of deep learning, ensuring the accuracy and reliability of neural network models is paramount. A key concept in achieving this is the idea of *Error-Revealing Capability*. This refers to the potential of a test instance to highlight or expose flaws and inaccuracies in a model. Rather than being a static measure, this capability is dynamic and quantifiable, typically assessed through mathematical scoring functions. These functions evaluate test instances based on criteria such as deviation from expected patterns, complexity, or other model-specific factors. Understanding and quantifying the error-revealing capability of each test instance is crucial, as it directly informs the process of *Test Prioritization*.

Test Prioritization in deep learning is the strategic process of identifying and ordering test instances based on their likelihood to reveal errors in the model. It involves leveraging the assessed error-revealing capabilities of these instances to make informed decisions about which ones to label and evaluate first. This approach is particularly beneficial when resources for labeling are limited. By prioritizing test instances that are most likely to uncover potential weaknesses in the neural network, this method ensures that the available resources are utilized most effectively. Moreover, this enhances the overall robustness and reliability of the neural network system, as it allows for the early detection and rectification of critical issues that could compromise model performance.

Test prioritization in deep learning involves several key components. The first component is the deep neural network, represented by $\mathcal{N}$, which takes an input x and produces an output y. This network is then challenged with a test dataset, comprising a set of unlabeled test instances $X_s = \{x_1, x_2, \ldots, x_n\}$, where n is the total number of test instances. A critical constraint in this context is the labeling cost, which allows for only m test instances to be labeled, where $m < n$. The second component is a scoring function, denoted by F_s, used to quantify the error-revealing capability of each test instance. Common scoring functions such as Gini Impurity or Entropy are often used for this purpose. The ultimate goal is to identify *error-revealing instances*, which are the test instances most likely to be misclassified by the neural network. By connecting these components, the process of test prioritization seeks to uncover the most significant weaknesses in the model with limited resources.

The formal representation of this process involves partitioning the test instances into two ordered sets based on their likelihood of revealing errors or being correctly classified by $\mathcal{N}$. This partitioning is defined by the equations:

$$X_e = \{(x_i, F_s(x_i)) \mid \mathcal{N}(x_i) \neq y_i, i \in [1,k], x_i \in X_s\} \tag{2.1}$$

$$X_c = \{(x_i, F_s(x_i)) \mid \mathcal{N}(x_i) = y_i, i \in [k+1,n], x_i \in X_s\} \tag{2.2}$$

In these equations, X_e contains all test instances that are likely to reveal errors in the model, indexed between 1 and k, and are prioritized for labeling. For any two instances x_i and x_j in X_e where $i, j \in [1,k]$ and $i < j$, it holds that $F_s(x_i) \geq F_s(x_j)$. This implies that $F_s(x_1)$, with the highest score, indicates the greatest likelihood of revealing an error, whereas $F_s(x_k)$ has the lowest score in X_e. Consequently, X_e is prioritized for labeling by human annotators to evaluate and improve the neural network.

On the other hand, X_c as defined in Eq. (2.2), includes all instances that are likely to be correctly classified, extending from the $(k+1)$th to the nth instance in X_s. Due to the constraint $m = k$, there are no resources left to label instances in X_c, thus focusing the labeling effort on the potentially error-revealing instances in X_e. Through this structured prioritization, the process effectively utilizes limited labeling resources to identify and address the most informative and critical instances, enhancing the neural network's accuracy and robustness.

2.7 RUNTIME MONITORING

Runtime monitoring in deep learning refers to the process of continuously observing and analyzing the internal operations of a neural network during its active use. This practice involves tracking the behavior of hidden layers and neurons to identify any unusual or unexpected activities. These are activities that deviate from what was observed during the model's training phase. Recently, various runtime monitors [28, 83] have been proposed. For instance, in our work, we follow the framework in [83], three-verdict monitors supervise how the neural network predicts the inputs and judge the network prediction with acceptance, uncertainty, or rejection verdicts. In the following, we recall how to construct and execute a monitor. For more details, we refer the reader to [28, 83].

2.7.1 *Monitor Construction*

After training, for each training instance, we collect the high-level features from the penultimate layer and the corresponding predicted class. Let $\mathbf{watch}(x, L_k)$ be a function that reads the output of L_k. The output of $\mathbf{watch}(x, L_k)$ is a $|L_k|$ dimension vector, denoted as $\vec{\mathbf{v}}$. We consider a training dataset as a set of subsets such that $X_t = \{X_1, X_2, ..., X_g\}$ where g is the total number of classes in a dataset. All instances in each subset have the same ground truth class. We also have a corresponding V_i for each subset X_i, where $V_i = \{\mathbf{watch}(x, L_{q-1}) = \vec{\mathbf{v}} \mid x \in X_i\}$.

After high-level features collection, we partition V_i into two subsets V_i^c, and V_i^{inc} according to whether a neural network correctly or incorrectly classifies the input, where V_i^c, and V_i^{inc} are formally defined as follows:

$$V_i^c = \{\mathbf{watch}(x, L_{q-1}) = \vec{\mathbf{v}} \quad | \quad x \in X_i, \mathcal{N}(x) = y_i\}$$

$$V_i^{inc} = \{\textbf{watch}(x, L_{q-1}) = \vec{\mathbf{v}} \quad | \quad x \in X_i, \mathcal{N}(x) \neq y_i\}$$

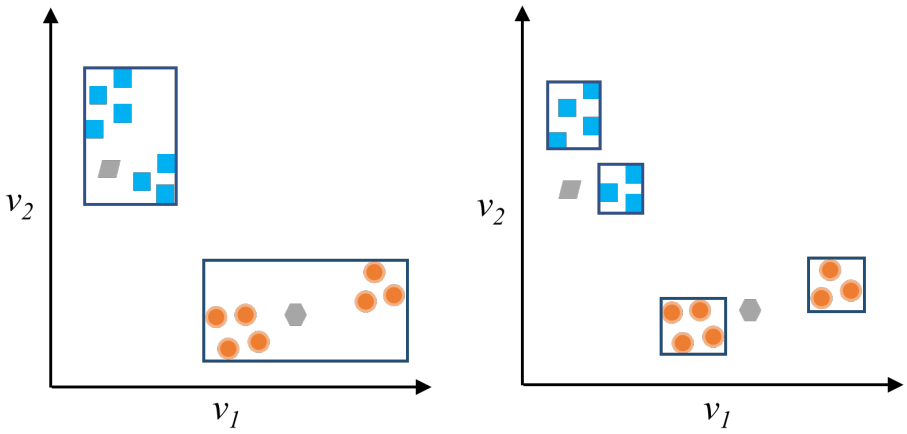
To construct box abstraction, assume $|\vec{\mathbf{v}}| = n$, $|V_i^c| = p$, and $|V_i^{inc}| = s$. Accordingly, there are two types of boxes B_i^c and, B_i^{inc} defined as follows:

$$B_i^c = \{[a_j, b_j] \mid 1 \leq j \leq n,\ a_j = \min_{k=1}^{p} V_i^c[j,k],\ b_j = \max_{k=1}^{p} V_i^c[j,k]\}$$

$$B_i^{inc} = \{[a_j, b_j] \mid 1 \leq j \leq n,\ a_j = \min_{k=1}^{s} V_i^{inc}[j,k],\ b_j = \max_{k=1}^{s} V_i^{inc}[j,k]\}$$

Box abstraction or *minimum bounding box* is a union of a list of intervals such that $B_i = \{[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]\}$ [83]. The underlying assumption is that instances of the same class show a similar pattern because they are more contiguous in the feature space than instances of other classes. Thus, monitors have two types of boxes for each class: one box contains correctly classified instances, and the other box contains incorrectly classified instances.

When a novel input is slightly similar to other instances inside the box, monitors falsely accept the network prediction. For example, in Fig. 2.5 (a), there are two classes of squares and circles, and the novel inputs are parallelogram and hexagon, respectively. The neural network incorrectly classifies the novel inputs as square and circle, respectively. We can obtain more accurate verdicts from monitors by clustering all instances into small boxes. In Fig. 2.5 (b), monitors correctly reject the predictions of the neural network for the novel inputs because the novel instances are outside the boxes. Thus, the process of clustering should occur before the construction of boxes. The clustering process has a hyperparameter τ, which controls the size of each box abstraction, i.e., the smaller the τ value is, the more compact the box abstraction is and vice versa.



(a) Without clustering (b) With clustering

Figure 2.5: Novel test instances before and after clustering.

2.7.2 *Monitors Execution*

During the evaluation of the neural network, if a test instance is inside one of the correctly classified boxes, the monitor accepts the prediction. If a test instance is inside an incorrectly

classified box, the monitors reject the prediction. The verdict of monitors is uncertain if the test instance is in an overlapping region between the correctly and incorrectly classified boxes. Lastly, if the test instance is outside all boxes, the monitors reject the network prediction. In the last case, monitors consider this instance a novel input [49] or Out-Of-Distribution (OOD) [13].

2.8 STATISTICAL SCORING FUNCTIONS

The scoring function aims to estimate the error-revealing capability of each test instance and score it accordingly. We present two scoring functions, namely *impurity* measures: Gini Impurity and entropy. They are heavily used in statistics [17, 55] and machine learning fields [8]. We first provide an overview of a decision tree. Then, we define and explain the properties of each measure and discuss how they are mainly used as a splitting criterion in the context of decision tree [60]. Finally, we illustrate how impurity measures are reused as a scoring function in the test prioritization area.

2.8.1 *Decision tree*

Decision tree [5] is a supervised machine learning algorithm used mainly as a classifier. Intuitively, it consists of a hierarchical mapping of nodes. Each node comes in the form of an if-else statement. A tree starts with a root node, which contains all instances in the dataset. The root node is split into children nodes based on a specific impurity criterion. The splitting process continues recursively until the tree reaches the leaves where the impurity score is zero, i.e., all instances in the node belong to a certain class.

2.8.2 *Gini Impurity or Gini Index*

Gini Impurity or Gini Index (GI) [59] is a measure of the impurity in the distribution of classes over the node. GI is widely used in decision trees [70] by which the tree can decide the best feature that splits the current node into more pure nodes. Hence, a feature with a lower GI score is selected. Moreover, GI is used in various applications as an uncertainty measure, e.g., it has been utilized recently to measure the degree of certainty to predict the pixel depth to construct a 3D point cloud [38]. We calculate the GI of the node as follows:

$$GI = 1 - \sum_{i=1}^{C} p_i^2 \tag{2.3}$$

where p_i is the probability of an instance being classified to class i in a node and $\sum_{i=1}^{C} p_i = 1$.

PROPERTY 1 Gini Impurity value is within the interval, $[0, 1[$ i.e., zero indicates the purity of a node and in that case, all instances of the node belong to the same class. On the other side, when the value approaches 1, it indicates maximum heterogeneity among classes in a node, i.e., the node has at least one instance from each class.

PROPERTY 2 If all classes have the same probability in a node, Gini Impurity reaches the maximum value:

$$\begin{aligned} GI_{max} &= 1 - C * (1/C)^2 \\ &= 1 - (1/C) \end{aligned} \tag{2.4}$$

Assume we have a C classification problem where C is a very large number e.g., greater than 1000 classes, and all classes have the same number of instances. Hence, Gini Impurity approaches 1. This case is rare in real-life classification problems because of the difficulty of finding many classes having the same probability in the network output, i.e., an image equally shares some features from each class.

EXAMPLE 3: *As illustrated in Fig. 2.6, given a binary classification problem, and $C = 2$. Gini Impurity is 0.5 which is the peak of the curve when there is an equal number of instances for each class, i.e., the probability of each class is 0.5.*

2.8.3 *Entropy or Shanon entropy*

Entropy [69] is a metric used widely in information theory to measure the amount of randomness or variability in the data being processed. Entropy is heavily used in machine learning problems, e.g., decision tree frequently utilizes entropy as a branching criterion [70]. The value of entropy is computed by the following equation:

$$entropy = -\sum_{i=1}^{C} p_i \ \mathbf{log_2}(p_i) \tag{2.5}$$

where p_i is the probability of an instance being classified to class i in a node and $\sum_{i=1}^{C} p_i = 1$.

PROPERTY 3 Entropy ranges between 0 to n where $n \in \mathbb{N}$ and computed by Eq. 2.6. An entropy value of 0 indicates a completely pure tree node, meaning it perfectly classifies the instances. Conversely, an entropy value of n suggests complete impurity, indicating that all classes in the node have an equal probability of classification.

PROPERTY 4 Maximum entropy, denoted as $entropy_{max}$, is achieved when the probability distribution across classes is uniform, i.e., $p_1 = p_2 = \ldots = p_C = 1/C$. In such a case, the entropy reaches its peak value n. This maximum entropy is mathematically represented as $\log_2(C)$, as shown in the equation:

$$\begin{aligned} entropy_{max} &= -C * (1/C) * \mathbf{log_2}(1/C) \\ &= \mathbf{log_2}(C) \end{aligned} \tag{2.6}$$

EXAMPLE 4: *Figure 2.6 illustrates how the entropy changes over the changes of classification probability in a binary classification problem. The maximum value of entropy is 1 when the probability of each class is the same.*

The first work [6] published in test prioritization selected the entropy value as a scoring function to prioritize buggy-revealing test instances. Likewise, DeepGini [22] employed Gini Impurity as a measure for the likelihood of misclassification for test instances. Following the same approaches, we use Gini Impurity and entropy as statistical scoring functions in conjunction with the monitors' verdicts to prioritize the incorrectly predicted instances among the unlabeled test dataset.

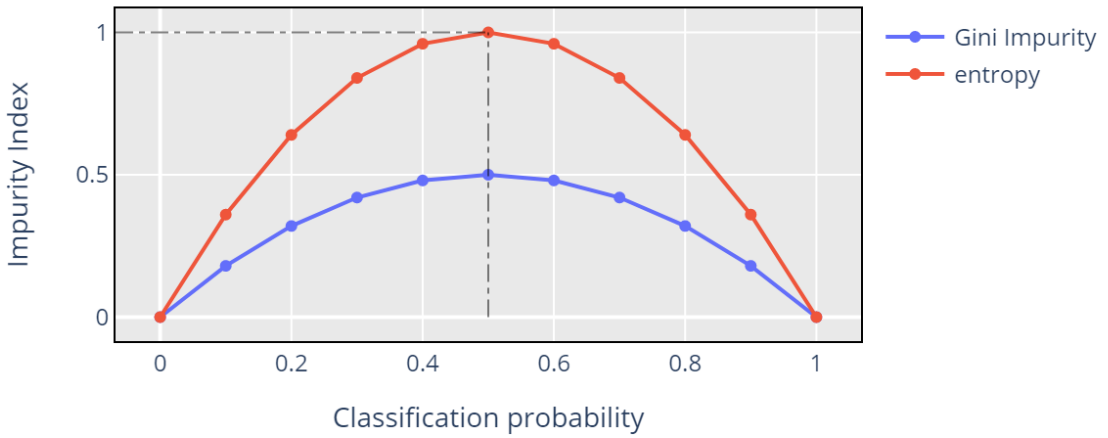


Figure 2.6: Impurity metrics in binary classification problem.

EXAMPLE 5: *Assume we have a binary classification with two classes {●, ■}. We also have 6 unlabeled test instances with the network output. As demonstrated in table 2.1, the Gini Impurity and entropy have similar behaviors in measuring the uncertainty of the neural network towards the test instances. We can observe that instance D has the highest impurity score (Gini Index = 0.5, entropy = 1.0) which is more likely to be misclassified by the network. The prioritization list contains other instances C, F, E, A, and B which are descendingly prioritized by both impurity scores as shown in the last column. Table 2.1 intuitively shows that both metrics have the same impact on test prioritization.*

Table 2.1: Prioritizing Test Data Using the Gini Index (GI)

Instance	Ground truth	DNN output	Predicted class	Gini index	Entropy	Order
A	●	0.90 , 0.10	●	0.18	0.47	5
B	■	0.00 , 1.00	■	0.00	0.00	6
C	●	0.40 , 0.60	■	0.48	0.97	2
D	■	0.50 , 0.50	●	0.50	1.00	1
E	■	0.25 , 0.75	■	0.38	0.81	4
F	●	0.35 , 0.65	■	0.46	0.93	3

2.9 EVALUATION METRIC

This section introduces an essential background to understand the existing metrics used to evaluate the test prioritization algorithms. Besides, we define each metric with its equation. We also explain the drawbacks of each metric through some scenarios.

2.9.1 *APFD*

The Average Percentage of Faults Detection (APFD) is a metric that primarily evaluates the performance of test prioritization algorithms in the software testing domain [21]. We compute the APFD value by the following equation:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{mn} + \frac{1}{2n} \tag{2.7}$$

where n is the total number of test cases, and m is the number of faults exposed in the test dataset. TF_i is the order of a test case that exposes the fault(i).

The APFD value ranges between 0 and 1. When APFD value is zero, the test prioritization algorithm works ineffectively, i.e., the order of the test cases which expose faults (TP_i) at the end of the test dataset and vice versa. For instance, if the number of faults is 10 in the 10000-test dataset. The order of the test cases that expose the faults is the last ten between 9990 and 10000. The APFD is almost zero, which indicates the slow faults detection rate.

Since there is no labeling for test cases in software testing, the APFD metric completely ignores the labeling budget over its calculation. In the following example, we discuss how ineffectively APFD evaluates the prioritized test dataset when we add the labeling budget.

EXAMPLE 6: *Assume we have an unlabeled test dataset that contains 1000 test cases. The dataset has 100 error-revealing test cases that are prioritized between 101 and 200. The APFD value is calculated as follows:*

$$\begin{aligned} APFD_1 &= 1 - \frac{\sum_{i=1}^{100} TF_i}{mn} + \frac{1}{2n} \\ &= 1 - \frac{101 + 102 + ... + 199 + 200}{100 * 1000} + \frac{1}{2 * 1000} \\ &= 0.85 \end{aligned}$$

Within the labeling budget (100), the correct value of APFD should be *zero* because all the first 100 test cases are not exposing faults. Therefore, the APFD over-evaluates erroneously the performance of the test prioritization algorithm from 0.0 to 0.85.

2.9.2 *RAUC*

The Ratio Area Under Curve (RAUC) is an evaluation metric defined as the ratio between the area under the curve of the actual performance to the area under the ideal curve. We compute the RAUC value by the following formula:

$$RAUC = \frac{\sum_{i=1}^{m} Practical_i}{\sum_{i=1}^{m} Ideal_i} * 100\% \tag{2.8}$$

where m is the labeling budget. The RAUC ratio is between 0% and 100%, where 0% indicates that the test prioritization algorithm does not prioritize any error-revealing test cases within the labeling budget and vice versa.

The main issue with the RAUC metric is that it evaluates the prioritization algorithm on how quickly an algorithm prioritizes the faults, namely, *fault detection rate*. The Fault Detection RatE (FDRE) should not be the only factor to evaluate the performance of the

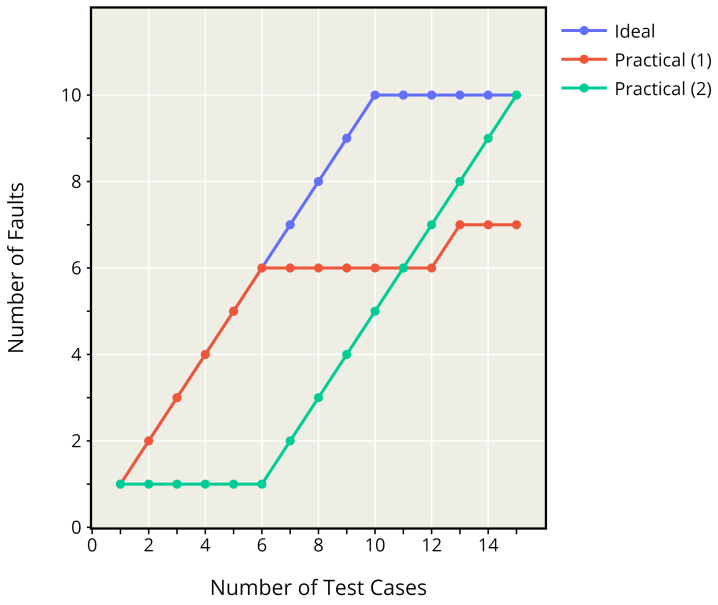


Figure 2.7: RAUC metric evaluates 2 test prioritization algorithms: Practical (1) & Practical (2)

prioritization algorithms. But the metric should also involve how many erroneous test cases the algorithm prioritizes, namely, Fault Detection RatiO (FDRO). In the following example, we demonstrate the importance of the second factor.

EXAMPLE 7: *Assume we have a 100-unlabeled test dataset that has 10 error-revealing test cases. The labeling budget is 15 test cases. Figure 2.7 shows the order of the test cases prioritized by the algorithm on the x-axis. We have two different prioritization algorithms: Practical 1 and Practical 2. The former algorithm has a high fault detection rate, and the latter one detects more faults. We calculate the RAUC for both algorithms as follows:*

$$RAUC_1 = \frac{\sum_{i=1}^{15} Practical(1)_i}{\sum_{i=1}^{15} Ideal_i} * 100\%$$

$$= \frac{1+2+3+4+5+6*7+7*3}{1+2+3+4+5+6+7+8+9+10*6} = \frac{78}{105} = 74.29\%$$

$$RAUC_2 = \frac{\sum_{i=1}^{15} Practical(2)_i}{\sum_{i=1}^{15} Ideal_i} * 100\%$$

$$= \frac{1*6+2+3+4+5+6+7+8+9+10}{1+2+3+4+5+6+7+8+9+10*6} = \frac{60}{105} = 57.14\%$$

$RAUC_1$ shows that the first algorithm prioritizes the first six test cases that expose faults quickly. On the other hand, $RAUC_2$ shows that the second algorithm starts poorly prioritizing test cases. But after the 6th test case, all the test cases, ranked between 7 and 15, are error-revealing test cases in the test dataset. Hence, the latter algorithm has a higher fault detection ratio than the former one. However, the RAUC metric incorrectly evaluates the first prioritization algorithm as more effective with a significant margin reaching up

17.15%. Therefore, the evaluation of RAUC is misleading, as illustrated in the previous example, and should be corrected.

2.9.3 *ATRC*

The Average Test Relative Coverage (ATRC) is formulated by TestRank[39]. The ATRC metric involves the labeling budget in the calculation. We compute the ATRC value by the following formula:

$$
\begin{aligned}
TRC &= \frac{\text{\#Detected Faults}}{\min(\text{\#Labeling Budget, \#Total Faults})} \\
ATRC &= \frac{1}{m}\sum_{i=1}^{m} TRC_i * 100\%
\end{aligned}
\tag{2.9}
$$

where Test Relative Coverage (TRC) is the ratio between the number of detected faults to the minimum of the labeling budget and the total number of faults in the test dataset. The ATRC is the average of TRC when the labeling budget (m) is less than or equal to the total number of faults in the dataset. The ATRC metric is more effective than the APFD metric since the ATRC evaluates the performance of an algorithm under a limited budget rather than the entire dataset. In other words, the ATRC metric is a stress test within a limited budget. The following example demonstrates how the ATRC metric behaves over two prioritized datasets: (i) Dataset A has a high fault detection rate, and (ii) Dataset B has a high fault detection ratio.

EXAMPLE 8: *Assume we have a 100-unlabeled test dataset. There are ten faults in the test dataset, and the labeling budget is 30. Moreover, two algorithms prioritize the test dataset, which results in two prioritized datasets, namely A and B. Dataset A is [1, 1, 1, 1, 1, 1, 0, 0, 0, 0], and dataset B is [1, 0, 0, 1, 1, 1, 1, 1, 1, 1] where one represents faults, and zero represents non-faults. We compute the ATRC for both datasets as follows:*

$$
ATRC_1 = \frac{1}{10}\sum_{i=1}^{10} TRC_i * 100\% = \frac{1}{10} * \left[\frac{1}{1} + \frac{2}{2} + \frac{3}{3} + \frac{4}{4} + \frac{5}{5} + \frac{6}{6} + \frac{6}{7} + \frac{6}{8} + \frac{6}{9} + \frac{6}{10}\right] = 88.74\%
$$

$$
ATRC_2 = \frac{1}{10}\sum_{i=1}^{10} TRC_i * 100\% = \frac{1}{10} * \left[\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{2}{4} + \frac{3}{5} + \frac{4}{6} + \frac{5}{7} + \frac{6}{8} + \frac{7}{9} + \frac{8}{10}\right] = 66.42\%
$$

We should calculate the ATRC value when the labeling budget is less than or equal to the number of faults (10). We can see that the ATRC metric suffers from the same problem as the RAUC metric. More specifically, the ATRC metric evaluates an algorithm only on the fault detection rate, not the ratio. Thus, the *ATRC* metric evaluates the first and second prioritization algorithms with 88.74%, and 66.42%, respectively. We can also observe clearly how the ATRC metric falsely over-evaluates the first algorithm over the second algorithm, with a significant difference (up to 22.32%). We can conclude that the current metrics are ineffective and have drawbacks. Thus, the evaluation metrics should give more weight to the fault detection ratio over the fault detection rate. If two algorithms have the same fault detection ratio, the algorithm with the high fault detection rate should be superior.

2.10 ACTIVE LEARNING

This section introduces the concept of active learning and its role in training deep learning models. It also details its mathematical representation as an iterative optimization problem. This section concludes with a comparative analysis.

2.10.1 *Overview*

Active learning [84] is a semi-supervised learning technique where the learning algorithm actively queries an oracle (such as a human expert) to obtain labels for specific examples from a pool of unlabeled data. The goal is to select the most informative examples that, once labeled, will contribute the most to improving the model's performance.

In the context of deep learning with unlabeled *training* data, active learning can be particularly beneficial. Since labeling data can be time-consuming and expensive, especially in complex domains, active learning aims to minimize the number of labels needed by intelligently selecting the most *informative* samples.

EXAMPLE 9: *In an active learning framework applied to image classification, consider a neural network model, denoted as f, developed to classify various classes. The model initially operates on a dataset D that comprises a limited number of labeled images. A much larger dataset U consists of unlabeled images.*

As part of the active learning strategy, the model evaluates each image in the unlabeled dataset U using an informativeness measure $I(x; f)$. This measure assesses the potential contribution of each image x in U towards improving the model's learning process. Key approaches [19] to this measure include Query-by-Committee, where samples are selected based on label disagreements among multiple classifiers. Also, Max-Margin sampling focuses on samples with maximum uncertainty. Lastly, Max-Entropy sampling uses entropy to estimate the uncertainty for each image x in U. These methods efficiently guide the labeling process to enhance the model's performance during the training process.

The most informative images, forming a subset S from U, are then labeled – either by domain experts or through automated methods – and added to the labeled dataset D. The model f is subsequently retrained on this enriched dataset. By iteratively repeating this process, the model incrementally improves its ability for classification. This process exemplifies the core principle of active learning: selectively augmenting the training dataset with the most impactful examples to optimize the learning process.

2.10.2 *Active Learning vs. Test Prioritization*

Table 2.2 illustrates that active learning and test prioritization are two distinct methodologies in deep learning. Active learning concentrates on training models efficiently by selecting the most informative examples actively. On the other hand, test prioritization focuses on improving the testing process by prioritizing the most critical or likely-to-fail tests. The table compares these two aspects based on several criteria such as purpose, main focus, and key benefits. It also considers the involvement of human experts and the iterative process, highlighting their distinct roles in enhancing the efficiency and effectiveness of deep learning model training and testing.

Table 2.2: Comparison between Active Learning and Test Prioritization in Deep Learning.

Aspect	Active Learning	Test Prioritization
Purpose	To efficiently train models by actively selecting the most informative examples.	To efficiently test models by prioritizing the execution of the most critical or likely-to-fail tests.
Main Focus	Training phase of the model.	Testing phase of the model.
Key Benefit	Reduces the need for large labeled datasets.	Reduces the time and computational cost of labeling and inspection.
Iterative Process	Typically involves iterative selection, labeling, and retraining.	Typically involves iterative prioritization, labeling, and testing and debugging.

2.11 TEST SELECTION

This section begins with an overview that defines the test selection area. Then, we provide a comparative analysis, elucidating the unique roles and objectives of test selection and test prioritization in enhancing deep learning model testing.

2.11.1 *Overview*

Test selection [43] is the process of choosing a subset of test cases from a larger test suite to validate a model's performance. In deep learning, test selection aims to identify the most *informative and representative* test cases that are likely to evaluate specific functionalities of the model. Unlike test prioritization, which focuses on ordering the test cases, test selection is about choosing a subset that maximizes the effectiveness of testing within resource constraints (e.g., time, and computational resources).

2.11.2 *Test Selection vs. Test Prioritization*

Table 2.3 illustrates that test selection and test prioritization are two distinct areas in deep learning testing. While test selection focuses on choosing a subset of test cases that maximize testing effectiveness within resource constraints, Test prioritization aims to order the test cases to increase the likelihood of early fault detection. The table compares these two aspects based on their purpose, main focus, objective, and mathematical formulation, shedding light on their unique roles in enhancing the efficiency and effectiveness of deep learning model testing.

Table 2.3: Comparison between Test Selection and Test Prioritization in Deep Learning.

Aspect	Test Selection	Test Prioritization
Purpose	To choose a subset of test cases that maximizes testing effectiveness.	To order test cases to increase the likelihood of early fault detection.
Main Focus	Selecting relevant test cases.	Ordering the execution of error-revealing test cases.
Objective	Maximize effectiveness within resource constraints.	Maximize fault detection rate and ratio within resource constraints.
Order	Unordered test instances	Ordered test instances

2.12 CONCLUSION

In conclusion, this chapter has combined the essential principles and techniques that form the backbone of the research. It has offered a thorough overview that connects various domains, from the mechanisms of DNNs to the subtle strategies involved in test prioritization. By shedding light on statistical scoring functions, evaluation metrics, and closely related areas such as active learning, and test selection. The chapter serves as both a guide and a gateway, setting the stage for the deeper investigation that lies ahead.

Part II

CONTRIBUTIONS

3

DEEPABSTRACTION: 2-LEVEL PRIORITIZATION FOR UNLABELED TEST INPUTS IN DEEP NEURAL NETWORKS

Main contributions of this chapter

- We conduct the first study investigating the effectiveness of monitors in the test prioritization area for deep learning systems.
- We introduce a comprehensive study of the regions of the misclassified instances and the regions where the neural network can misclassify the instances with high confidence.
- We achieve state-of-the-art performance, demonstrated by empirically comparing our framework with other test prioritization techniques.

In this chapter, we begin by formalizing the test prioritization problem and then introduce our proposed solution. Our algorithm is divided into two main phases. First, *testing part* emphasizes the data processing and monitoring operations. Second, *prioritization algorithm* outlines the strategy for optimally ranking test instances considering the labeling budget. We then explain the effect of removing zero-scored instances in prioritization through an example. Then, we provide a practical example that showcases how our methodology efficiently detects misclassified instances. After that, we detail our experimental setup and the research questions. Lastly, we address the research questions from different perspectives: the effectiveness, efficiency, and stability of framework performance.

3.1 PROBLEM FORMULATION

Given a trained neural network $\mathcal{N}$ with a labeled training dataset X_t and an unlabeled test dataset X_s. We consider $|X_s| = n$ and the labeling cost is enough for only m test instances where $m \ll n$. We can utilize the test prioritization techniques to prioritize more error-exposing instances. Assuming there are k error-revealing instances in the test dataset and the cost of labeling is m which is only enough to label k test instances. Then, the ideal prioritization algorithm groups the test instances into two ordered sets as follows:

$$\mathrm{X_e} = \{F_s(x_i) \quad | \quad \mathcal{N}(x_i) \neq y_i, i \in [1,k], x \in \mathrm{X_s}\} \tag{3.1}$$

$$\mathrm{X_c} = \{F_s(x_i) \quad | \quad \mathcal{N}(x_i) = y_i, i \in [k+1,n], x \in \mathrm{X_s}\} \tag{3.2}$$

where $\mathrm{F_s}$ is a scoring function either Gini Impurity or entropy. The scoring function $\mathrm{F_s}$ helps in ranking the instances in $\mathrm{X_s}$ such that the instance with the highest likelihood of being misclassified gets the maximum score, and the one with the least likelihood gets the minimum score. The objective function we aim to optimize is Maximize $\sum_{i=1}^{k} F_s(x_i)$, where $F_s(x_i)$ is the score assigned by the scoring function to the i^{th} test instance in X_e. This objective function aims to maximize the identification of error-revealing instances within the budget m.

Ideally, eq. (3.1) shows that $\mathrm{X_e}$ contains all error-revealing test instances indexed between 1 and k. More specifically, for any $i, j \in [1,k]$ and $i < j$, then $\mathrm{F_s}(x_i) \geq \mathrm{F_s}(x_j)$. In addition, $\mathrm{F_s}(x_1)$ has the maximum score and $\mathrm{F_s}(x_k)$ has the smallest score in X_e. Eventually, X_e is further labeled by human annotators to evaluate a neural network. While eq.(3.2) shows that $\mathrm{X_c}$ includes all correctly classified test instances. There is no cost left to label $\mathrm{X_c}$ since the labeling budget has been run out on the k test instances. For clarity, Table 3.1 summarizes the notations used in this section.

Table 3.1: Summary of Notations

Symbol	Description
$\mathcal{N}$	The trained neural network
X_t	Labeled training dataset
X_s	Unlabeled test dataset
n	Size of X_s
m	Labeling budget
k	Number of error-revealing test instances
F_s	Scoring function, e.g., Gini Index, Entropy

We assume that the neural network $\mathcal{N}$ is fully trained and will not undergo further training after the test instances are labeled. This is a critical assumption as the error-revealing capability of test instances remains static. In a scenario where $\mathcal{N}$ undergoes further training or updates, the composition of X_e and X_c might change. This change thereby affects the effectiveness of the test prioritization algorithm and the subsequent evaluation of $\mathcal{N}$.

3.2 ALGORITHM

The algorithm operates in two main phases: the training phase and the testing phase as shown in Fig. 3.1. During the training phase, the algorithm extracts high-level features and categorizes vectors according to their predicted class. These vectors then play a pivotal role in constructing the boxes of the monitors. This phase sets the stage for the testing phase, where the algorithm employs a hierarchical ranking system.

During testing, one can view DeepAbstraction as a hierarchical ranking system with two distinct levels. Initially, it segments the test instances into three categories depending on the monitor verdict. It then ranks instances within each category according to scores from a scoring function. Annotators subsequently label these ranked instances, staying within the predefined labeling budget. We will further detail the primary stages of our framework in algorithm 1. It is worth noting that the algorithm is designed to be flexible. One can easily swap out the scoring function or introduce additional categories based on the monitor verdict, allowing for easy adaptability to different testing scenarios.

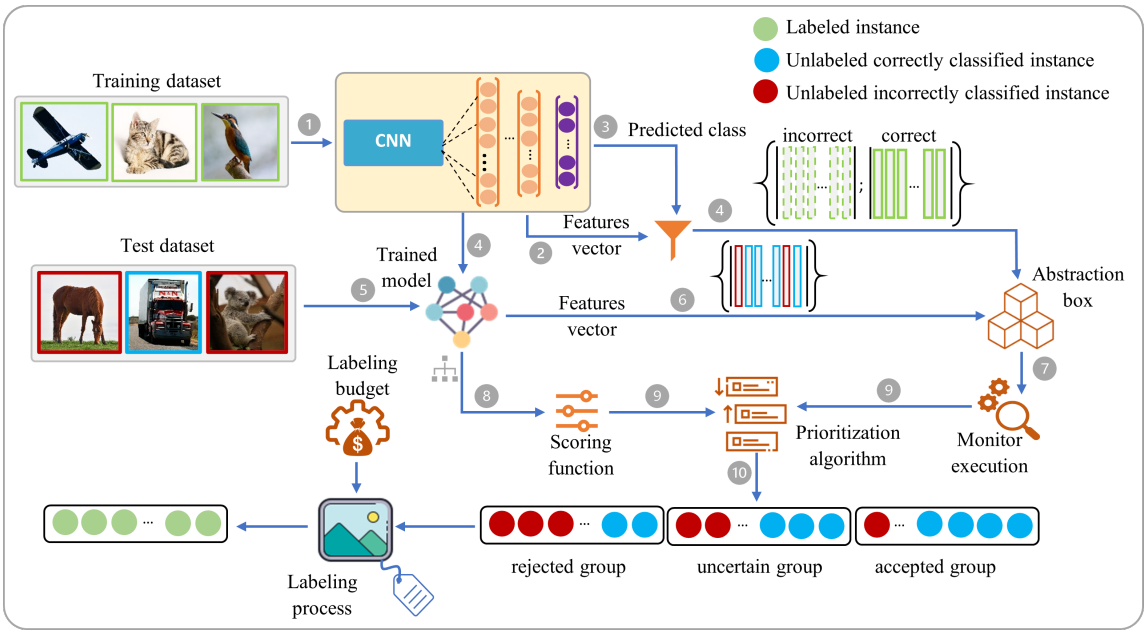


Figure 3.1: DeepAbstraction architecture and workflow.

3.2.1 *Testing part*

The algorithm 1 uses an index-based approach for efficient data processing. This is particularly useful for large test datasets, as it speeds up the lookup and retrieval operations (line 3). Then, we extract the high-level features vector for each test input and read the corresponding predicted class y' (lines 4-5). On line 6, the softmax function determines the classification probability of each class $output$. We then choose one of the two statistical scoring methods, either Gini Impurity or entropy, to evaluate the score of each test instance based on eq. (3.4) and (3.3), as outlined in line 7.

Next, we analyze the location of the current vector, $\vec{\mathbf{v}}$, within the feature space, and get the verdict accordingly. To conclude the process, we record the index of the instance

and its corresponding Gini or entropy score in the verdict dictionary, D, as shown in lines 9-14. This dictionary serves as a data structure for quick reference, enabling efficient prioritization and subsequent labeling. Finally, the design of the algorithm allows for easy integration with other machine learning pipelines, making it a versatile tool for various test prioritization tasks.

3.2.2 *Prioritization algorithm*

We organize the unlabeled test instances into three distinct categories based on the monitor's verdict: rejected, uncertain (or suspicious), and accepted. In this sequence, we expect that the majority of misclassified instances will belong to the *rejected* category. The *uncertain* category contains a smaller number of misclassified instances, while the *accepted* category has the fewest. Table 3.2 summarizes the presence of both correctly and incorrectly classified instances in each of these groups. The instances in each group are in random order. Furthermore, the primary objective of test prioritization is to prioritize the misclassified instances or the error-revealing instances. This means highlighting the true positives in the *rejected* group, the uncertain positives in the *uncertain* group, and the false negatives in the *accepted* group.

As the network classifies most instances accurately with ease, several instances receive a Gini Index and entropy score of zero. These zero-scored instances typically represent false positives, uncertain negatives, and true negatives in each group. For budgetary reasons, each group should ideally exclude such instances. This means that to optimize the budget, it is beneficial to eliminate false positives from the *rejected* category and focus on labeling more uncertain positives in the *uncertain* category. Likewise, omitting zero-scored instances from the *uncertain* category can enhance the labeling of false negatives in the *accepted* group (line 19). In essence, by eliminating zero-scored instances, which are typically easy for the model to classify, the algorithm ensures that the labeling budget is spent on instances that are genuinely challenging for the model. We will analyze empirically the efficacy of this step over different benchmarks in 3.3.1.

The final step involves ranking the instances based on their scores and selecting a subset that aligns with the available labeling budget (as detailed in lines 20-23). This step is crucial as it determines which instances will actually be reviewed by human annotators. The effectiveness of the algorithm in this step directly impacts the quality of the labeled dataset and, by extension, the performance of the neural network in real-world applications.

3.2.3 *Example*

In the upcoming example, we highlight the significance of the zero-removal step within the DeepAbstraction workflow. Our main goal is to label misclassified instances, focusing on true positives (TP), uncertain positives (UP), and false negatives (FN) across all groups. Consider a scenario where we have a fixed budget that allows for labeling only 50 test instances. Within each of our three groups - rejected, uncertain, and accepted - there are 50 instances. In the rejected group, 30 instances are true positives (TP) and 20 are false positives (FP). Notably, 18 out of these 20 FPs have a score of zero. By eliminating these 18 zero-scored FPs from the rejected group, we decrease the number of instances we need to label in that group to 32. The strategic removal enhances our labeling process by making it more efficient. Additionally, it gives priority to other misclassified instances, such as

Algorithm 1: Test prioritization algorithm

Input: $X_{test} = \{x_1, x_2, ..., x_n\}, B^c, B^{inc}$
$\mathcal{N}$: DNN, *bdgt* : *labeling budget*
scoring_method : either Gini Impurity or entropy
Output: $X_{prioritized} : \{x_1, ..., x_{bdgt}\}$

```
/* Monitors execution */
D ← {}
foreach x ∈ X_test do
    idx ← index(x)
    v⃗ ← extract(x)
    y′ ← classify(N, x)
    output ← softmax(v⃗)
    score ← score_calculation(output, scoring_method)
    if (v⃗ ∈ B^c[y′]) ∧ (v⃗ ∈ B^inc[y′]) then
        D[uncertain].append([idx, score])
    else if v⃗ ∈ B^c[y′] then
        D[accepted].append([idx, score])
    else
        D[rejected].append([idx, score])
/* Prioritization algorithm */
sorted_indx ← [ ]
prioritized_indx ← [ ]
verdicts ← [rejected, uncertain, accepted]
foreach i ∈ verdicts do
    D[i] ← remove_zero(D[i])
    indices_lst ← sort(D[i])
    sorted_indx.extend(indxs_lst)
prioritized_indx ← prioritize(sorted_indx, bdgt)
X_prioritized ← X_test[prioritized_indx]
return X_prioritized
```

Table 3.2: Verdict types and definitions over all groups.

Verdict Type	Definition
True Positive (TP)	monitors truly reject the incorrect prediction
False Positive (FP)	monitors falsely reject the correct prediction
Uncertain Positive (UP)	monitors are uncertain towards incorrect prediction
Uncertain Negative (UN)	monitors are uncertain towards correct prediction
False Negative (FP)	monitors falsely accept the incorrect prediction
True Negative (TN)	monitors truly accept the correct prediction

UPs and FNs, in the following groups. Consequently, we label the top 18 instances in the uncertain group, which are more likely to be uncertain positives. In essence, by refraining from expending our budget on zero-scored FPs in the rejected group, we can reallocate resources to label UPs in the uncertain group. Likewise, by removing zero-scored instances in the uncertain group, we can redirect our efforts toward labeling false negatives in the accepted group.

The example demonstrates the power of strategic budget reallocation. By removing zero-scored instances from the *rejected* group, we not only save resources but also free up the budget to focus on more challenging instances in the *uncertain* and *accepted* groups. This reallocation is crucial for maximizing the utility of a limited labeling budget.

3.3 EXPERIMENTAL SETUP

In the first two sections, we discuss the datasets and models that form the basis of our experiments. We then detail the evaluation metric in the 3.3.4 section, followed by a comparison of our framework against state-of-the-art baselines in the 3.3.3 section. Finally, in the 3.3.5 section, we highlight the primary questions guiding our research.

3.3.1 *Datasets*

- **MNIST**: the MNIST (Modified National Institute of Standards and Technology database) is a seminal dataset in the machine learning community, often referred to as the "hello world" of image classification. It Comprises 70,000 grayscale images of handwritten digits from 0 to 9 as shown in Fig.3.2. Each image is standardized to a size of 28x28 pixels. These images are further divided into two subsets: a training set with 60,000 images and a test set consisting of 10,000 images [15].

Figure 3.2: Samples from MNIST dataset.

- **Fashion-MNIST**: the Fashion-MNIST dataset, developed as a more challenging alternative to the classic MNIST dataset. It serves as a modern benchmark for machine learning algorithms in the domain of computer vision. This dataset consists of 70,000 grayscale images, each of 28x28 pixel resolution, representing ten categories of clothing and accessories as shown in Fig.3.3. The images are systematically divided into two subsets: a training set comprised of 60,000 images and a testing set of 10,000 images [85].

Figure 3.3: Samples from FMNIST dataset.

- **CIFAR-10**: the CIFAR-10 dataset, introduced by the Canadian Institute for Advanced Research, stands as a pivotal benchmark in computer vision research. This dataset comprises 60,000 color images, uniformly distributed across ten distinct classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks as illustrated in Fig.3.4. Each image is rendered in a 32x32 pixel resolution and is encapsulated in a three-channel RGB format. For a structured evaluation of machine learning models, the dataset is divided into a training subset with 50,000 images and a test subset that consists of 10,000 images [34].

Figure 3.4: Samples from CIFAR-10 dataset.

- **SVHN**: the Street View House Numbers (SVHN) dataset, sourced from Google's Street View imagery, serves as a complex benchmark in the domain of computer vision research. This dataset encompasses over 600,000 color images, capturing house numbers embedded in a multitude of real-world settings, delineated by diverse fonts, colors, and potential occlusions as illustrated in Fig.3.5. Each image is articulated in the RGB color. Uniquely, SVHN emphasizes the recognition of multi-digit numbers, diverging from the more foundational single-digit tasks characteristic of datasets such as MNIST. The dataset is systematically divided into three subsets: a training set with 73,257 digits, a testing set with 26,032 digits, and an extra set containing 531,131 digits that can be optionally used for training [46].

Figure 3.5: Samples from SVHN dataset.

3.3.2 *DNN Models*

- **LeNet-5**: represents a foundational CNN architecture tailored for character recognition. The model consists of seven layers, alternating between convolutional and average pooling layers, followed by two fully connected layers and a softmax classifier. This structure was pioneering, allowing the extraction of hierarchical spatial features from input images. Its success on the MNIST dataset for digit recognition

underscored the potential of CNNs in computer vision tasks, laying the groundwork for subsequent architectural advancements.

- **VGG16**: is a deep CNN architecture known for its homogeneity, primarily using 3x3 convolutional layers stacked in increasing depth. Its design is both straightforward and deep, resulting in compelling accuracy on the ImageNet dataset. Its depth and simplicity made it a benchmark for image classification tasks and a base for various transfer learning applications[36].
- **ResNet18**: is a variant of the Residual Network (ResNet) architecture, characterized by its unique *skip connections* or *residual blocks*. These connections allow gradients to flow through the network directly, mitigating the vanishing gradient problem in deep networks. ResNet18, with its 18 layers, achieved remarkable performance on ImageNet. It demonstrates the efficacy of deeper networks without the traditional associated training difficulties[27].
- **GoogLeNet**: introduces the inception module, a novel multi-scale architecture allowing the network to adapt to various spatial dimensions of features. Notably, GoogLeNet achieved top performance in the ImageNet Large Scale Visual Recognition Challenge. It utilizes fewer parameters than many contemporary models, emphasizing efficiency along with accuracy[74].
- **AlexNet**: is renowned for significantly advancing the field of deep learning in computer vision. This deep CNN, with its five convolutional and three fully connected layers, demonstrated a notable reduction in error rates on the ImageNet competition[35].

3.3.3 *Baselines*

- Entropy[6]: is utilized as a metric to summarize the output distribution of a neural network, especially the softmax output. Entropy quantifies the uncertainty or randomness in a distribution. The paper defines the entropy of a distribution with the following formula:

$$H(p) = -\sum_{c=1}^{C} p_c \log p_c \tag{3.3}$$

  where C represents the number of output classes and p_c denotes the predicted probability for class c. In the context of the softmax output, a higher entropy value indicates a more spread-out or uncertain classification, where the predicted probabilities are more uniformly distributed across classes. On the other hand, a lower entropy value suggests a more certain classification, where one predicted probability notably surpasses the others. This entropy-based measure assists in assigning priority scores to inputs based on their uncertainty, providing insights into the network's confidence levels.
- Distance-based Surprise Adequacy (DSA)[6]: is a method to assess the surprise or novelty of an input to a neural network relative to known inputs. This is achieved by comparing the activation traces of the new input with those from known inputs. An activation trace is defined as the vector of activation values observed when classifying

an input, representing how each neuron in the neural network is activated. For the DSA computation, the closest neighbor of a new input x with the same predicted class is identified as x_a. Subsequently, another input x_b is found, which is closest to x_a but has a different predicted class. The distances *dista* and *distb* are then computed based on the differences in their activation traces. The underlying rationale of DSA is that a more surprising input might more likely reveal an erroneous behavior in the trained model, as it indicates the model may not be well-prepared for such an input.

- DeepGini[22]: is a metric introduced to prioritize testing data for DNNs to enhance their quality. The foundation of DeepGini is rooted in a statistical perspective of DNNs, specifically focusing on the likelihood of misclassification of a given input. The DeepGini metric for a test t with a DNN output vector $[pt_1, pt_2, \ldots, pt_N]$, where the probabilities sum up to 1 (i.e., $\sum_{i=1}^{N} pt_i = 1$), is defined as:

$$\xi(t) = 1 - \sum_{i=1}^{N} pt_i^2 \tag{3.4}$$

  In this equation:

  - $\xi(t)$ is the DeepGini score for test t.
  - N represents the number of output classes.
  - pt_i denotes the predicted probability of the test t belonging to the i^{th} class.

  A higher value of $\xi(t)$ indicates that the probabilities across classes are more evenly distributed, suggesting a higher likelihood of misclassification. Conversely, a lower score implies that one class has a significantly higher predicted probability, indicating more certainty in the classification. In the context of test prioritization, a higher DeepGini score means that the test is more "surprising" or "novel" to the DNN, and thus, it should be prioritized higher. This is because such tests can potentially reveal areas where the DNN might be uncertain or prone to errors.

- TestRank[39]: TestRank evaluates each unlabeled test data point using two key attributes: intrinsic and contextual. The intrinsic attributes are the direct output responses the model provides for a given input, such as the predictive output distribution. In contrast, the contextual attributes offer insights into the model's behavior by analyzing the classification accuracy of similar, already labeled samples. By constructing a similarity graph that encompasses both labeled and unlabeled instances, TestRank leverages graph-based semi-supervised learning to extract these contextual insights. Combining both sets of attributes, TestRank calculates a metric that predicts the potential of a test instance to reveal a model failure. Consequently, test instances are ranked and prioritized, ensuring that the most uncertain or risky ones are addressed first.

3.3.4 *Evaluation metrics*

In this section, we introduce two metrics to evaluate our framework. The first is the ATRC [39], which evaluates the effectiveness of test prioritization approaches. The second is a new metric, called distance ratio, by which we estimate the regions where DeepGini and DeepAbstraction are more effective in the feature space.

3.3.4.1 *Average Test Relative Coverage (ATRC)*

We adopt the evaluation metric ATRC, as outlined in TestRank [39]. This metric is particularly suitable because ATRC takes into account the labeling budget. Conversely, many methods, as seen in [6, 22], utilize the APFD[56] as the primary evaluation criterion. While APFD is predominantly used in software test prioritization, it does not consider the labeling cost. Given that there is no labeling expense in software test prioritization, using APFD in scenarios that require considering labeling costs becomes inapplicable. We compute the ATRC value by the following formula:

$$\begin{aligned} TRC &= \frac{\text{\#Detected Faults}}{\min(\text{\#Labeling Budget, \#Total Faults})} \\ ATRC &= \frac{1}{m}\sum_{i=1}^{m} TRC_i * 100\% \end{aligned} \tag{3.5}$$

where TRC is the ratio between the number of detected faults to the minimum of the labeling budget and the total number of faults in the test dataset. The ATRC is the average of TRC when the labeling budget (m) is less than or equal to the total number of faults in the dataset.

3.3.4.2 *Distance-based Metric*

We introduce the distance-based metric to better understand the positioning of misclassified instances. This metric aims to determine whether a misclassified instance is located closer to the centroid of its class or to the classification boundary. Firstly, let's define our set of centroids. Assume $C = \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2, ..., \vec{\mathbf{c}}_g$, where g represents the total number of classes present in the dataset. Next, let u denote the number of instances associated with a particular class. Correspondingly, c stands for the centroid vector of that specific class. Lastly, consider $\vec{\mathbf{v}}_i$, which signifies the high-level features we have extracted from the penultimate layer of our model. We can now compute the centroid of a certain class with the following formula:

$$\vec{\mathbf{c}} = \frac{\sum_{i=1}^{u} \vec{\mathbf{v}}_i}{u} \tag{3.6}$$

We define **a distance ratio** as a ratio of the Euclidean distance between an instance and the centroid of the predicted class y' to the Euclidean distance between an instance and the centroid of the ground truth y, defined formally as follows:

$$d(x) = \frac{\|\vec{\mathbf{c}}_{y'} - \vec{\mathbf{v}}\|_2}{\|\vec{\mathbf{c}}_{y} - \vec{\mathbf{v}}\|_2} \tag{3.7}$$

Based on the distance ratio, we group misclassified instances into three main regions:

1. If $d(x) \in]0.0, 0.7]$, a misclassified instance becomes a **near-centroid** instance.
2. If $d(x) \in [0.7, 1.3]$, a misclassified instance is a **near-boundary** instance.
3. If $d(x) \in]1.3, \infty]$, a misclassified instance becomes also a **near-centroid** instance.

The difference between the first and third regions is that, in region 1, an instance is close to the centroid of the incorrectly predicted class. In contrast, in region 3, an instance is marginally close to the centroid of the ground-truth class where the neural network still misclassifies this instance.

Since defining an accurate decision-boundary region in feature space is often complicated, we need to estimate the near-boundary region. We found through extensive experimental studies that this region is approximately defined by a distance ratio of 1.0 ± 0.3. Note that when the distance ratio is 1, instances are more likely to be very close to the decision boundary.

3.3.5 *Research Questions*

We empirically evaluate DeepAbstraction from three perspectives: effectiveness, efficiency, and performance stability. We answer the following questions:

- **RQ1 (Effectiveness)**: How does the effectiveness of DeepAbstraction in prioritizing error-revealing instances compare to other deep learning test prioritization methods?
- **RQ2 (Efficiency)**: How efficient is DeepAbstraction in terms of time complexity and reducing labeling costs?
- **RQ3 (Stability)**: How does the clustering parameter τ influence the performance stability of DeepAbstraction?

3.4 EXPERIMENTAL EVALUATION

In this section, we address these questions and provide an in-depth discussion of the results.

3.4.1 *RQ1: Effectiveness*

To answer **RQ1**, we make the following comparisons: first, we compare DeepAbstraction with DSA, Entropy, and DeepGini. Then, due to the particular data split of TestRank, we compare DeepAbstraction separately to TestRank (we follow the same dataset splits).

Table 3.3 overviews the effectiveness comparison between DeepAbstraction and the other baselines. Overall, DeepAbstraction outperforms other approaches significantly in all datasets. DeepAbstraction is much more effective than DSA, e.g., experiment (D, FMNIST) in Table 3.3 shows an ATRC improvement of 39.15%. Moreover, our framework achieves better results than Entropy and DeepGini, with a considerable margin ranging between $11.54\% \sim 30.11\%$ in terms of ATRC.

As shown in Table 3.4, our framework remarkably outperforms TestRank in 4 out of 6 experiments. Additionally, another advantage of our framework over TestRank is that our framework avoids the pre-labeling efforts in TestRank. For instance, TestRank in experiments (A, CIFAR10) and (C, SVHN) pre-labeled 8,000 and 10,000 test instances to operate, respectively, which is very time-consuming. The comparison in Table 3.4 is unfair since TestRank requires more budget, i.e., pre-labeling cost. If we count this pre-labeling cost into the budget of TestRank, we find out that DeepAbstraction is considerably superior to TestRank in all datasets.

Table 3.3: Comparing the effectiveness of DeepAbstraction with other baselines based on ATRC(%).

Exp. ID	Dataset	DSA	Entropy	DeepGini	DeepAbstr.	Δ
A	MNIST	33.15	34.67	50.36	**61.90**	↑+11.54
B	MNIST	27.84	29.44	33.98	**57.58**	↑+23.60
C	F-MNIST	41.77	42.59	56.23	**86.34**	↑+30.11
D	F-MNIST	43.66	41.21	55.27	**82.81**	↑+27.54
E	CIFAR-10	43.13	42.72	57.17	**79.12**	↑+21.95
F	CIFAR-10	47.98	52.73	60.20	**83.10**	↑+22.90
G	SVHN	47.44	50.73	59.92	**83.60**	↑+23.68
H	SVHN	47.22	44.54	55.06	**83.82**	↑+28.76

Table 3.4: Comparing the effectiveness of TestRank and DeepAbstration in terms of ATRC(%).

Exp. ID	Model	Dataset	Validation Acc. (%)	TestRank (%)	DeepAbstr. (%)	Δ
A	ResNet-18	CIFAR 10	70.10	87.87	84.15	↓- 03.72
B	ResNet-18	CIFAR 10	66.40	85.53	89.31	↑+03.78
C	ResNet-18	CIFAR 10	68.30	76.56	85.02	↑+08.46
D	Wide-ResNet	SVHN	94.20	76.36	85.91	↑+09.55
E	Wide-ResNet	SVHN	92.50	66.06	83.19	↑+17.13
F	Wide-ResNet	SVHN	81.60	95.32	86.89	↓- 08.43
Total						↑+26.77

Next, we investigate why our framework is more effective than other techniques in two aspects: i) the regions where other methods fail to prioritize error-revealing instances; ii) the removal of zero-scored instances inside each group, as mentioned in line 19 of algorithm 1.

Regarding the regions, Table 3.5 shows that there is a significant number of error-revealing instances residing close to the centroids in column *total*. DeepGini and Entropy can poorly estimate the error-revealing capability of test instances residing in regions 1 and 3 (near-centroid regions), as defined in 3.3.4. On the contrary, DeepAbstraction works effectively in all regions, as demonstrated in almost all experiments in Table 3.5. For instance, the results of experiment E show that DeepGini and DeepAbstraction prioritize approximately the same number of near-boundaries instances (371 versus 373), respectively. However, DeepAbstraction prioritizes 202 near-centroid instances highly greater than the 65 instances prioritized by DeepGini.

As for the removal of zero-scored instances, Fig. 3.6 shows that this step reduces considerably the number of false positives (defined in Table 3.2), particularly in FMNIST and MNIST datasets. Although, there is a slight drop in the number of true positives. This

Table 3.5: The number of instances DeepAbstraction and DeepGini detect in each region.

Exp. ID	Dataset	Near-Boundary instances			Near-Centroid instances		
		total	DeepAbst.	DeepGini	total	DeepAbst.	DeepGini
A	MNIST	92	**56**	49	54	**40**	13
B	MNIST	31	**28**	14	26	**16**	6
C	F-MNIST	450	**303**	229	415	**306**	191
D	F-MNIST	478	**312**	301	275	**212**	72
E	CIFAR-10	630	**373**	371	267	**202**	65
F	CIFAR-10	1017	**639**	631	440	**309**	113
G	SVHN	1319	**842**	736	444	**269**	171
H	SVHN	760	464	**501**	405	**344**	50

drop is due to the GI function weakness that scores some true positives with zero scores, e.g., these instances in regions 1 and 3. In a nutshell, the removal process for zero-scored instances improves the effectiveness of DeepAbstraction by prioritizing more uncertain positives from the second group.

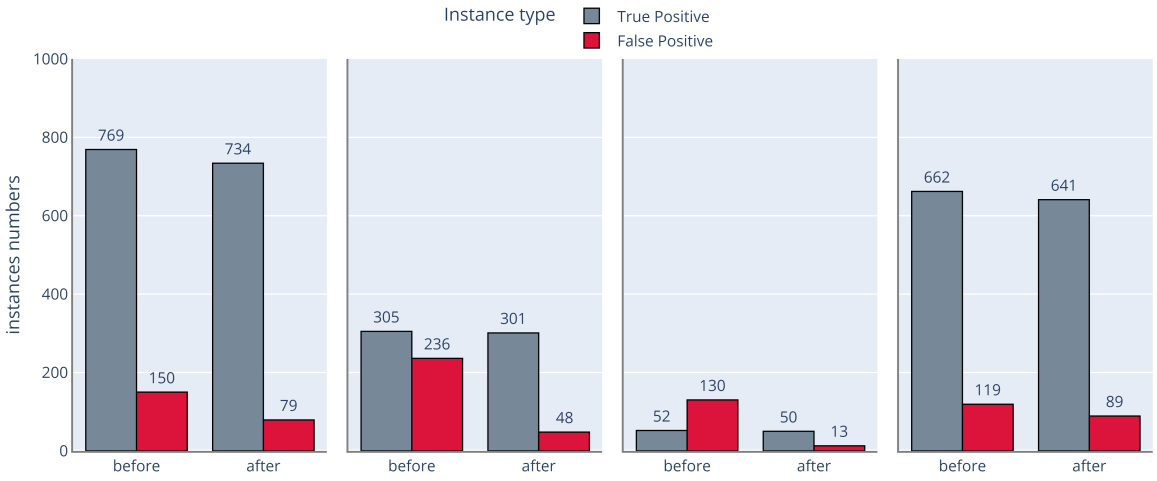


Figure 3.6: The effect of removing zero-scored instances.

RQ1 Answer :

DeepAbstraction emerges as a more effective test prioritization technique compared to other state-of-the-art methods, including TestRank. Unlike existing methods, DeepAbstraction can accurately estimate the error-revealing capability of test instances across all regions, including near-centroid instances that are often challenging to classify correctly.

3.4.2 *RQ2: Efficiency*

DeepAbstraction consists of two main components: monitors and prioritization algorithm. According to [83], the complexity of monitors construction is the complexity of the clustering $\mathcal{O}(m^2)$ where m is a number of training samples. The complexity of the membership query and scoring function for test instances is $\mathcal{O}(n)$ where n is the number of test instances. While the complexity of sorting algorithm is $\mathcal{O}(n \times \log n)$. Therefore, the overall complexity of DeepAbstraction is $\mathcal{O}(m^2)$. In practical terms, the time spent on clustering is minimal compared to the time required for labeling the entire dataset. This makes DeepAbstraction not only theoretically efficient but also practically advantageous.

DeepAbstraction outperforms TestRank in efficiency for three key reasons:

1. **Pre-Labeling Requirement**: TestRank needs to pre-label some test instances, adding to its time complexity.
2. **Complexity of kNN Graph**: TestRank uses a kNN graph for similarity measures, which has a complexity of $\mathcal{O}((m+n)^2)$.
3. **Neural Network Overheads**: TestRank employs two neural networks (Graph Neural Network and MLP), which significantly increase its time complexity.

RQ2 Answer :

DeepAbstraction is an efficient framework and the time complexity is $\mathcal{O}(m^2)$. The time complexity of DeepAbstraction is negligible compared to the time of manual labeling. In practice, DeepAbsraction is much more efficient than TestRank.

3.4.3 *RQ3: Stability*

In this section, we delve into the stability of DeepAbstraction's performance. We focus on two key aspects: the impact of the hyperparameter τ on performance across different benchmarks, and a comparative analysis of DeepAbstraction's stability against TestRank. The hyperparameter τ plays a crucial role in determining the compactness of the box abstraction. A smaller τ results in a more compact box, while a larger τ leads to a more coarse box.

We observed that when models have very high validation accuracy (greater than 98%), the box abstraction tends to be too compact. This compactness results in a high number of false positives during the testing phase. To mitigate this, we can adjust τ to a larger value, which allows the box to include more true negatives and fewer false positives. For example, a τ value of 0.4 provided stable ATRC results in experiments E and F as demonstrated in Fig. 3.7.

On the flip side, when dealing with models that have high accuracy but are not perfect, we encounter the issue of coarse boxes. These boxes lead to more false negatives during testing. To address this issue, we can reduce the τ value to make the box more compact, capturing more true positives and fewer false negatives. In our experiments, a τ value of 0.05 yielded more stable ATRC results in experiments A, B, C, and H.

In comparison to TestRank, the results in column 6 of Table 3.4 confirm the performance stability of DeepAbstraction among different datasets. Whereas column 5 shows that

TestRank has high unstable performance, e.g., although experiments E and F are conducted on the same dataset and model, the ATRC results differ significantly, 95.32% vs. 66.06%.

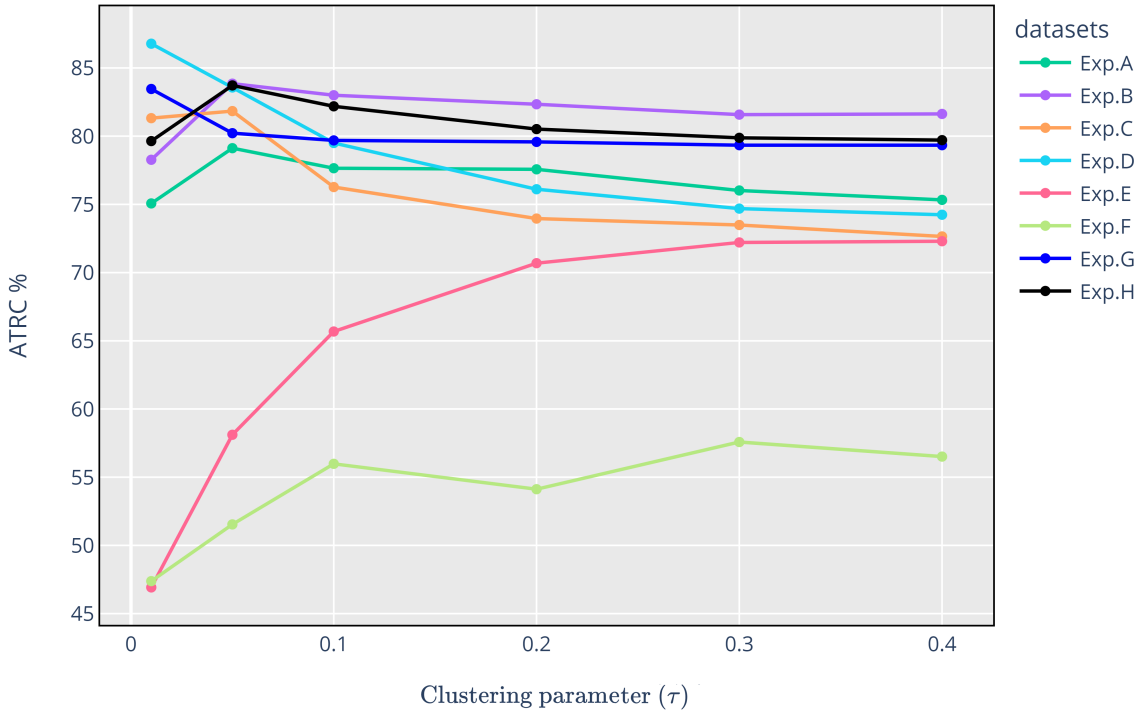


Figure 3.7: Impact of Clustering Parameter τ on Performance Stability in DeepAbstraction.

RQ3 Answer :

The default value of τ is 0.05 to provide stable performance in DeepAbstraction. With very highly accurate models, τ should be 0.4 to provide a stable performance. Furthermore, DeepAbstraction is more stable than the recent work (TestRank) on various benchmarks.

3.5 CONCLUSION

In this chapter, we introduced a practical 2-tier prioritization framework named *DeepAbstraction*, to estimate the error-revealing capability of unlabeled test instances. At its core, DeepAbstraction integrates two pivotal ranking mechanisms: monitors and a scoring function. Notably, monitors in DeepAbstraction help significantly to detect more error-exposing instances in both regions: near the centroids and the decision boundaries. Empirical evidence clearly demonstrates that DeepAbstraction surpasses other deep learning test prioritization algorithms, even outperforming the State-Of-The-Art (SOTA) algorithm (TestRank).

4

DIFFICULTY AND SEVERITY-ORIENTED METRICS FOR TEST PRIORITIZATION IN DEEP LEARNING SYSTEMS

Main contributions of this chapter

- We conduct the first intensive study to investigate the effectiveness of the existing evaluation metrics.
- We develop a novel metric (WFDR) that solves the limitations of the predecessors.
- We develop a new metric (SFDR) that evaluates algorithms in the context of severity prioritization.

In this chapter, we begin by discussing the Misclassification Ratio (MR), a key concept that sheds light on our first metric. We give a practical example to help grasp the importance of MR. This example shows how some metrics might overlook MR and how that can affect the evaluations. We unfold our proposed metrics: the WFDR and the SFDR. Following that, we detail our experimental setup, including the datasets, models, baselines, and the research questions. Lastly, we address the research questions from different perspectives: the effectiveness of metrics, the effectiveness of algorithms, and the distribution of the severity levels.

4.1 MISCLASSIFICATION RATIO

The MR is the ratio between the number of misclassified instances and the size of the test dataset, ranging from 0 to 1. A low ratio indicates that there are relatively few misclassified instances compared to the overall size of the dataset. This scarcity makes it difficult for the algorithm to prioritize these few instances, and vice versa. As a result, the difficulty of prioritization is inversely proportional to the misclassification ratio. As such, it is unfair to evaluate the algorithm's prioritization capability without considering the misclassification ratio. For instance, a 100-test dataset has 20 misclassified instances. The initial misclassification ratio is 20/100. If the algorithm has already prioritized the first 19 misclassified tests correctly, only one misclassified test remains. However, this last misclassified test is now hidden among 81 correctly classified tests. Finding and prioritizing this single misclassified test becomes much more challenging.

4.2 MOTIVATIONAL EXAMPLE

EXAMPLE 10: *Let's consider a hypothetical scenario to illustrate a limitation in current evaluation metrics. Imagine we have two datasets: Dataset A with 14 test instances and Dataset B with 1000 test instances. Both datasets have eight misclassified instances.*

An algorithm prioritizes the instances in both datasets as follows: [1,1,1,1,0,0,0,0], where 1 represents a misclassified instance and 0 represents a correctly classified one. The algorithm yields an RAUC of 77.22% and an ATRC of 81.73% for both datasets.

However, datasets A and B have different misclassification ratios, each metric evaluates the performance equally in both datasets. Intuitively, the algorithm performance in dataset B should be higher than in dataset A since the prioritization process is more difficult in B than in A. As a result, we need to develop an evaluation metric that considers the misclassification ratio.

4.3 WEIGHTED FAULTS DETECTION RATIO

The process of prioritization typically becomes more challenging as it progresses. Thus, it is necessary to assign weights at each step of the process. These weights should gradually increase with each successful step and decrease with each unsuccessful one. Accordingly, the last misclassified instance should have the largest weight. Therefore, we develop a new metric that involves the prioritization difficulty, called ***W**eighted **F**ault **D**etection **R**atio* (WFDR).

We compute the WFDR percentage by the following equation:

$$f(x) = \begin{cases} 1 & \text{if x is misclassified} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Actual} = \sum_{i=1}^{m} f(x_i) * \underbrace{\left[1 - \frac{m - d_{i-1}}{n - (i-1)}\right]}_{\text{Weights}} \tag{4.1}$$

$$\text{Ideal} = \sum_{i=1}^{m} \frac{n-m}{n-i+1} \tag{4.2}$$

$$\text{WFDR} = \frac{Actual}{Ideal} * 100\% \tag{4.3}$$

where m is the labeling budget, which equals the total number of faults, and n is the size of the test dataset. Also, d_i is the total number of the detected faults within the labeling budget (i). Under the ideal case, all faults are detected within the budget (i), hence, all $f(x_i) = 1$. Initially, no faults are detected, i.e., $d_0 = 0$.

EXAMPLE 11: *A 100-unlabeled test dataset has ten error-revealing test cases, and the labeling budget is 30. Moreover, two algorithms prioritize the test dataset, which results in two prioritized datasets, namely A and B. Dataset A is [1, 1, 1, 1, 1, 1, 0, 0, 0, 0], and dataset B is [1, 0, 0, 1, 1, 1, 1, 1, 1, 1] where one represents faults, and zero represents non-faults. The WFDR evaluates both approaches as follows:*

$$\begin{aligned} Actual_A &= \left[1 - \tfrac{10}{100}\right] + \left[1 - \tfrac{9}{99}\right] + \left[1 - \tfrac{8}{98}\right] + ... + 0 * \left[1 - \tfrac{4}{93}\right] + 0 * \left[1 - \tfrac{4}{92}\right] \\ &\quad + 0 * \left[1 - \tfrac{4}{91}\right] = 5.54 \\ Ideal_A &= 9.433, \qquad WFDR_A = 58.73\% \\ Actual_B &= 7.456,\ Ideal_B = 9.433,\ WFDR_B = 79.05\% \end{aligned}$$

Algorithm performance in dataset A reduces from 88.74% by ATRC and 81.82% by RAUC to 58.73% by WFDR. Likewise, the performance in dataset B increases from 69.09% by RAUC and 66.42% by ATRC to 79.05% by WFDR.

PROPERTY 1: *WFDR approaches its upper bound limit (FDRO) when the misclassification ratio is very small.*

Example 11 demonstrates that $FDRO_A$ is 60% and $FDRO_B$ is 80%. We see that the WFDR values are close to the FDRO values in datasets A and B, i.e., 58.78% and 79.05%, respectively. On the other hand, other metrics inaccurately over-evaluate the algorithm performance larger than the fault detection ratio. For example, the ATRC value in dataset A is 88.74% larger than 80%, and the RAUC value in dataset B is 69.09% larger than 60%.

In addition, we revisit example 11 to study how the misclassification ratio strongly affects the WFDR percentage. Figure 4.1 shows that the WFDR percentage approaches the FDRO percentage exponentially as the test dataset size increases from 14 (the minimum size of dataset A) to 200 and from 12 (the minimum size of dataset B) to 200. We conclude that as the dataset gets larger, the weights increase, and the prioritization difficulty becomes higher accordingly. In other words, each weight approaches one in eq. 4.1, and the sum of the weights is roughly the number of detected faults. In the ideal case, all $f(x_i) = 1$ in eq. 4.2, thus, the sum of all weights is roughly the total number of faults in the test dataset. From eq. 4.3, we obtain a WFDR value close to the FDRO, as shown in Fig. 4.1.

PROPERTY 2: *If two prioritized datasets have the same FDRO, the one with a higher FDRE has a greater WFDR.*

The property 2 holds when the misclassification ratio is a large value. When the misclassification ratio is small, the difference between the WFDR values of the two datasets diminishes drastically. For instance, if there are two prioritized datasets: A=[1,1,1,1,1,1,1,0,0,0] and B=[1,0,0,0,1,1,1,1,1,1] and the total number of faults in both datasets is constant (10). Figure 4.2 shows that dataset A significantly outperforms dataset B under small sizes of the test dataset, i.e., large misclassification ratios. The difference between the WFDRs of A & B reduces exponentially as the size of the test dataset increases. We justify property 2 by the weights change in the WFDR equation. The test dataset contains 13 instances: 10 misclassified and 3 incorrectly classified. We compute the WFDR for dataset B:

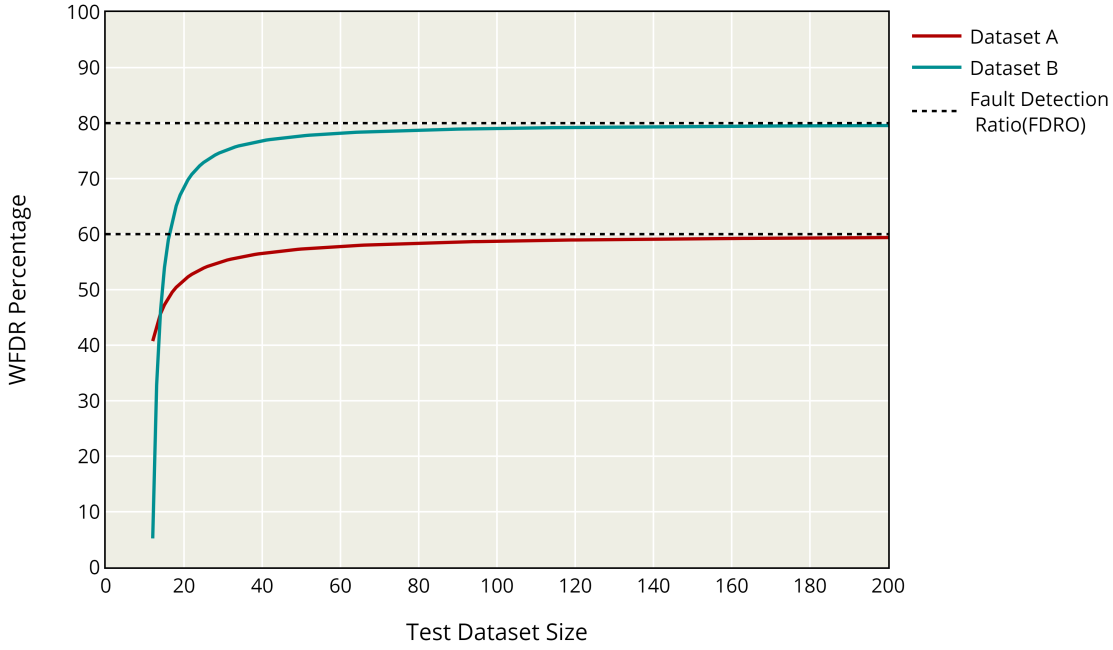


Figure 4.1: The effect of the dataset size on the WFDR under different FDROs.

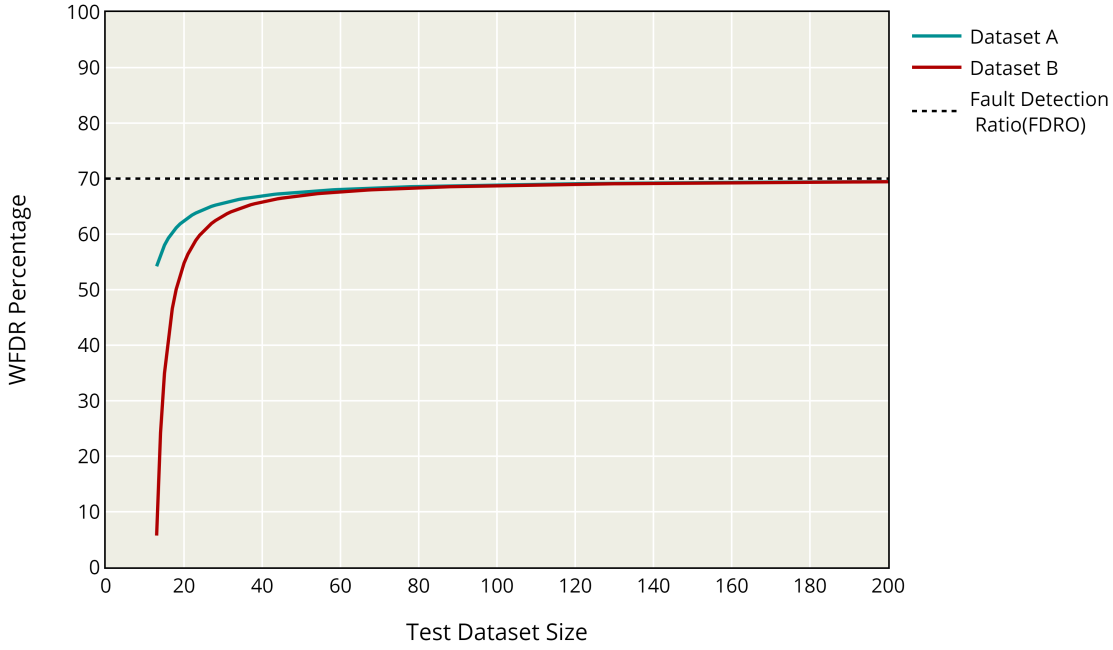


Figure 4.2: The effect of the dataset size on the WFDR under similar FDROs.

$$Actual_B = \left[1 - \tfrac{10}{13}\right] + 0 * \left[1 - \tfrac{9}{12}\right] + 0 * \left[1 - \tfrac{9}{11}\right] + ... + \left[1 - \tfrac{9}{9}\right] + \left[1 - \tfrac{8}{8}\right] + ... + \left[1 - \tfrac{4}{4}\right] = 0.231$$

$$Ideal_B = 4.04, \qquad WFDR_B = 5.72\%$$

$$Actual_A = 2.19, Ideal_A = 4.04, WFDR_A = 54.21\%$$

Since the algorithm in dataset B prioritizes all correctly classified instances, the remaining in the 13-test dataset has to be misclassified instances. Thus, there is no need for prioritization as the difficulty is zero, i.e., the weights (shown in blue) are zero. In dataset A, as the algorithm prioritizes the misclassified instances successfully, weights get larger. From Fig. 4.2, we can conclude that when the size of the test dataset size is:

- small (e.g., between 13 and 40): the MC is large, and the weights are small. Hence, the WFDR metric evaluates the algorithms depending on FDRE since both FDROs are the same.
- medium (e.g., 40 and 100): the MC tends to be low, and the weights and the difficulty get larger. Hence, there is much importance for FDRO.
- large (more than 100): both datasets have very small MC, and the difficulty is very high. Since both datasets have almost the same weights and difficulty, the WFDR evaluates the algorithms based on the FDRO rather than the FDRE.

4.4 SEVERE FAULTS DETECTION RATE

The existing algorithms prioritize all misclassified instances equally. In situations where the budget is limited or when safety and security are critical, it is insufficient to prioritize all misclassified instances similarly. Thus, the algorithm should consider the *severity* when prioritizing highly severe instances over other misclassified ones.

We estimate the severity level by the potential harm that can occur when the neural network misclassifies a particular instance. As a result, we quantify the severity by the prediction probability. For example, instances with a low-probability prediction of less than 50% are low-severity instances. Accordingly, we should plan safety precautions before model deployment to prevent damaging consequences. Contrarily, instances with a high prediction probability of more than 80% are considered highly severe. No proactive actions are taken since intelligent systems heavily rely on high-probability predictions.

Figure 4.3 illustrates some highly-severe examples from the CIFAR dataset. These instances reveal the main weaknesses of the trained neural network. As corrective actions, we should retrain the neural network with the following: a) boats with mainsail reflection, b) dogs at different zoom levels, and c) planes in various positions, not only flying.

In severity prioritization, algorithms should prioritize high-severity instances at the top of the list. In this context, the order among misclassified instances is more important. For example, prioritizing low-severity examples at the beginning negatively impacts the performance of the algorithm. The penalty is greater when the algorithm prioritizes correctly classified ones, which are completely safe. Therefore, the rate of prioritization is more important than the ratio.

An ideal list **A** has all misclassified instances in descending order according to the prediction probability. An algorithm prioritizes instances in a specific order in list **B**. To evaluate list **B** against **A**, we develop a new metric, namely ***S**evere **F**ault **D**etection **R**ate* (SFDR). We compute the SFDR percentage by the following equation:

$$f(x) = \begin{cases} 1 & \text{if x is correctly classified} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{SFDR} = \frac{1}{m} \sum_{i=1}^{m} \gamma^{f(x_i)} * \frac{|\mathbf{A}_{[0:i]} \cap \mathbf{B}_{[0:i]}|}{i} * 100\% \tag{4.4}$$

(a) label: ship
predicted: airplane
probability: 99.99%

(b) label: dog
predicted: horse
probability: 99.99%

(c) label: airplane
predicted: vehicle
probability: 99.70%

(d) label: airplane
predicted: vehicle
probability: 99.70%

Figure 4.3: High severe images in the CIFAR dataset.

where γ ranges $]0, 1[$, and γ controls the degree of the penalty. Moreover, the γ value and the penalty degree are inversely proportional. Thus, we select $\gamma = 0.5$ in our evaluations. Furthermore, the SFDR percentage ranges between 0% and 100% from worst to optimal performance, respectively. Lastly, the intersection operator allows for duplicated severe cases because some misclassified instances have the same severity degree. More importantly, the SFDR is a top-weighted metric, i.e., the metric imposes more weight on the top of the prioritized list. Thus, the cost of incorrect prioritization decays gradually along with the prioritized list.

We partition equally the degree of severity, i.e., the prediction probability, into 10 levels between 0% and 100%. For example, misclassified instances with a prediction probability of 90s% should be at the head of the prioritization list. On the other side, all correctly classified instances and misclassified instances with prediction probability between 1% and 10% should be at the tail of the list. It is worth noting that the prioritization among all misclassified instances of the same severity level does not matter, as illustrated by the following examples.

EXAMPLE 12: *An ideal list* $\boldsymbol{A}$ *= [99, 91, 85, 83, 64, 24],* $\boldsymbol{A}$ *contains all misclassified instances in a small test dataset, and the prioritized list* $\boldsymbol{B}$ *= [99, 0, 91, 83, 64, 0]. The labeling budget* $m = 10$ *and* $\gamma = 0.5$*. Since the severity of correctly classified instances is zero, we replace the prediction probability with zero. Let* $\boldsymbol{x}_i$ *be* $|\boldsymbol{A}_{[0:i]} \cap \boldsymbol{B}_{[0:i]}|$.

The first step to evaluate the prioritization list by SFDR metric is to encode the prediction probabilities into severity levels between 1 and 10, where 10 is the highest degree of severity (prediction probability between 90% and 100%) and 1 is the lowest degree of severity. More particularly, we replace all prediction probabilities with the corresponding level of severity. Thus, **A** becomes [10, 10, 9, 9, 7, 3] and **B** becomes [10,0,10,9,7,0].

Table 4.1 shows the step-by-step computation for the SFDR percentage. If the prioritized instance is correctly classified, the $f(x_i)$ value is one, and γ is 0.5. Thus, the SFDR value drastically declines when i is 2, and the value of $\mathbf{y}_i = \gamma^{f(x_i)} * (\mathbf{x}_i / i)$ largely decreases from 1 to 0.25. When i is 3, the intersection between the two lists allows duplication in the severity degree (10).

Table 4.2 demonstrates how the SFDR metric evaluates effectively different prioritization lists against **A** = [99, 90, 88, 81, 75, 70, 69, 62]. For example, in list **B**, the second incorrect prioritization heavily penalizes the performance with a significant drop from 100% to

Table 4.1: The SFDR metric evaluates the list **B**.

i	$\mathbf{A}_{[0:i]}$	$\mathbf{B}_{[0:i]}$	$\mathbf{x}_i$	$f(x_i)$	$\mathbf{y}_i$
1	[10]	[10]	1	0	1.00
2	[10, 10]	[10, 0]	1	1	0.25
3	[10, 10, 9]	[10, 0, 10]	2	0	0.67
4	[10, 10, 9, 9]	[10, 0, 10, 9]	3	0	0.75
5	[10, 10, 9, 9, 7]	[10, 0, 10, 9, 7]	4	0	0.80
6	[10, 10, 9, 9, 7, 3]	[10, 0, 10, 9, 7, 0]	4	1	0.33
$SFDR = \frac{1}{m}\sum_{i=1}^{m} \mathbf{y}_i * 100\%$					63.33%

80.09%. Moreover, the performance in **C** is worse than in **B** since the algorithm prioritizes falsely the last four instances that are non-fault instances. In scenarios **D** and **E**, the first four instances are incorrectly prioritized. The main difference between the two scenarios is the type of instances: misclassified instances with lower severity in scenario **D** and correctly classified instances in scenario **E**. As a result, there is an 18.28% drop in the SFDR percentage between the two scenarios. Note that calculations are not included for the sake of brevity, and only SFDR values are presented.

Table 4.2: SFDR evaluates different prioritization lists.

Name	List	SFDR (%)
B	[90, 88, 81, 75, 70, 69, 62, 99]	80.09
C	[99, 90, 88, 81, 0, 0, 0, 0]	65.86
D	[75, 70, 69, 62, 99, 90, 88, 81]	36.55
E	[0, 0, 0, 0, 99, 90, 88, 81]	18.27

4.5 EXPERIMENTAL SETUP

The experiments were conducted using a machine with an Nvidia K80 GPU and 12 GB of RAM, implemented using the PyTorch v1.9.0 framework. Table 4.3 provides a summary of the details of the main experiments. We detail our main setup as follows:

- **Datasets**: MNIST [15], Fashion-MNIST [85], CIFAR10 [34], SVHN [46].
- **Pretrained Model**: ResNet18 [27], GoogLeNet [74], ResNet34 [27], ResNet50 [27], ResNet101 [27], ResNet152 [27], and EfficientNet-B0 [75].
- **Prioritization Algorithms**: DeepGini, Neurons pattern, and DeepAbstraction.
- **Research Questions**:

  ❶ **(Metrics Effectiveness)**: How effective are the existing and proposed metrics in evaluating the prioritization algorithms?

❷ **(Algorithms Effectiveness)**: How effective are the existing algorithms evaluated by the WFDR and SFDR metrics?

❸ **(Severity Distribution)**: What is the distribution of highly severe instances among the widely used benchmarks?

Table 4.3: Details of the datasets and pretrained models.

Exp ID	Dataset	Training Dataset	Test Dataset	Pretained Model	Training Acc. (%)	Test Acc. (%)
Exp 1	CIFAR-10	50,000	10,000	Efficient-B0	94.95	92.86
Exp 2	CIFAR-10	50,000	10,000	ResNet101	88.83	86.97
Exp 3	F-MNIST	60000	10000	Efficient-B0	94.94	94.17
Exp 4	F-MNIST	60,000	10,000	ResNet50	93.11	91.12
Exp 5	MNIST	60,000	10,000	ResNet18	99.36	99.16
Exp 6	MNIST	60,000	10,000	ResNet34	99.29	98.84
Exp 7	SVHN	73,257	26,032	GoogLeNet	95.51	95.07
Exp 8	SVHN	73,257	26,032	ResNet152	94.63	94.10

4.6 EXPERIMENTAL EVALUATION

This section addresses the research questions outlined in section 4.5.

4.6.1 *RQ1: Metrics Effectiveness*

Figure 4.4 shows that the APFD metric overestimates the performance of all algorithms. For example, the APFD values for all experiments involving the DeepGini algorithm are between 94.14% and 99.64%. The APFD values are significantly larger than the other metrics (RAUC, ATRC) with differences up to 74%. Additionally, a comparison of APFD and ATRCwas conducted to understand the impact of the labeling budget. It was found that APFD values are greater than ATRCvalues with significant differences, reaching up to 47.19%, 82.91%, and 27.26% for each algorithm (a,b, and c). Since APFD does not consider the misclassification ratio, it incorrectly exceeds the evaluation of WFDR by a considerable margin of more than 70%.

RAUC and ATRCare not weighted metrics. Thus, they overestimate the performance, which exceeds the FDRO. For example, when comparing the evaluations of RAUC and WFDR for the DeepAbstraction algorithm among all experiments, the difference reaches 13.89%. Similarly, the difference between the evaluations of ATRC and WFDR reaches 20.04%, as shown in Fig. 4.4. Furthermore, RAUC and ATRCvalues among all experiments exceed the FDRO shown in Table 4.4. Contrarily, all values of the WFDR metric are either below or close to FDRO. For instance, in Exp 8, the FDRO of the DeepGini algorithm is 47.33%, while the RAUC, ARTC, and WFDR are 53.05%, 57.08%, and 46.55%, respectively.

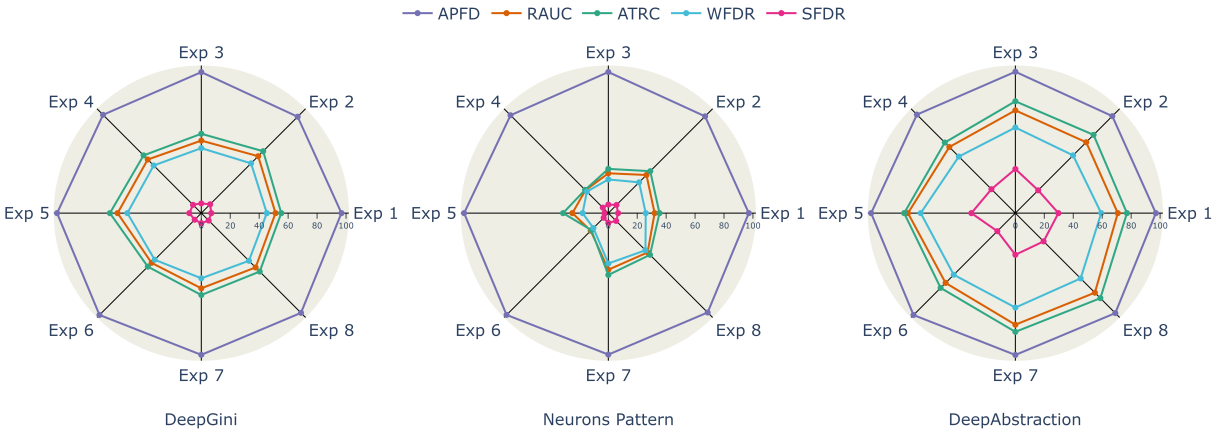


Figure 4.4: Evaluations of the different test prioritization metrics on various experiments.

Table 4.4: Fault detection ratio for several algorithms.

Exp ID	No. Bugs	Fault Detection Ratio (%)		
		DeepGini	Neurons Pattern	DeepAbstraction
Exp 1	714	46.22	26.61	60.22
Exp 2	1303	50.50	31.70	58.17
Exp 3	583	45.63	23.84	59.86
Exp 4	888	47.75	21.85	56.42
Exp 5	84	51.19	17.86	65.48
Exp 6	116	45.69	14.66	60.34
Exp 7	1284	45.64	35.44	65.81
Exp 8	1538	47.33	37.06	64.50

Lastly, Fig. 4.4 shows that SFDR evaluates the DeepGini algorithm as better than the neurons pattern algorithm. Table 4.5 shows the FDRO of two levels of severity: high level in which the prediction probability is greater than or equal to 80%, and moderate level in which the probability is between 50% and 80%. For instance, in Exp 5, 84 misclassified instances have different levels of severity: 36-high, 38-moderate, and 10-low. DeepGini prioritizes 0-high, and 33-moderate severity instances, while the neurons pattern algorithm prioritizes 2-high and 6-moderate severity instances. Hence, the SFDR values for DeepGini and neurons pattern algorithms are 8.3%, and 2.7%, respectively.

We can also observe that DeepAbstraction outperforms DeepGini, as the former algorithm prioritizes 25 highly severe instances, while the latter fails to prioritize any such instances. Nonetheless, DeepGini and DeepAbstraction prioritize 86.84%, and 18.42% of the moderately severe examples, respectively. As a result, the SFDR metric evaluates DeepAbstraction with (21.28%) as better than DeepGini with (8.05%). To sum up, the SFDR metric evaluation is consistent with the results in Table 4.5.

Table 4.5: Fault Detection Ratio (%) for different algorithms according to the severity levels.

Exp ID	FDRO of Severe Bugs				FDRO of Moderate Bugs			
	No. Bugs	Deep Gini	Pattern Algor.	Deep Abst.	No. Bugs	Deep Gini	Pattern Algor.	Deep Abst.
1	315	0.00	13.65	47.62	312	77.88	31.73	55.45
2	325	0.00	10.15	40.00	656	51.22	29.42	45.58
3	233	0.00	13.30	54.51	285	70.53	23.86	50.53
4	262	0.00	11.07	72.52	504	59.92	19.44	44.44
5	36	0.00	5.56	69.44	38	86.84	15.79	18.42
6	49	0.00	10.20	63.27	50	72.00	12.00	44.00
7	409	0.00	3.91	74.82	534	45.88	33.90	41.01
8	331	0.00	3.02	78.55	662	29.76	27.04	36.71

RQ 1 Answer :

The existing metrics are ineffective and also over-evaluate the performance more than the FDRO. On the other hand, the experiments show the validity of WFDR and SFDR evaluations.

4.6.2 *RQ2: Algorithms Effectiveness*

The answer to **RQ1** confirms that the proposed metrics effectively evaluate the performance of algorithms.

The evaluation of the WFDR metric shows that DeepAbstraction performs significantly better than other algorithms in all experiments, as demonstrated in Fig. 4.5. For example, DeepAbstraction outperforms DeepGini by a significant margin (up to 20.23%) as shown in Fig. 4.5. Since the WFDR metric relies heavily on the FDRO, the WFDR metric indicates that DeepGini performs better than the neurons pattern algorithm, which is consistent with the FDRO values in Table 4.4.

Figure 4.5 illustrates that all algorithms perform poorly in prioritizing highly severe instances, with SFDR values at most 30%. However, DeepAbstraction performs significantly better than other algorithms (DeepGini, Neurons Pattern), with margins of 23.8% and 27.7%, respectively.

The Gini score is inversely proportional to the certainty of the model, which is estimated by the prediction probability. DeepGini prioritizes high Gini instances with low certainty. i.e., low probability prediction. In other orders, low Gini instances with high prediction probability (high severity) are at the bottom of the priority list, resulting in poor performance in the severity prioritization. Table 4.5 confirms this by showing that DeepGini does not prioritize any highly severe instances with a prediction probability greater than 80%. But DeepGini prioritizes many moderate-severity instances.

The neurons pattern algorithm heavily relies on the Familiarity score (FD+) to measure the conformance degree between the established pattern during the training and the test

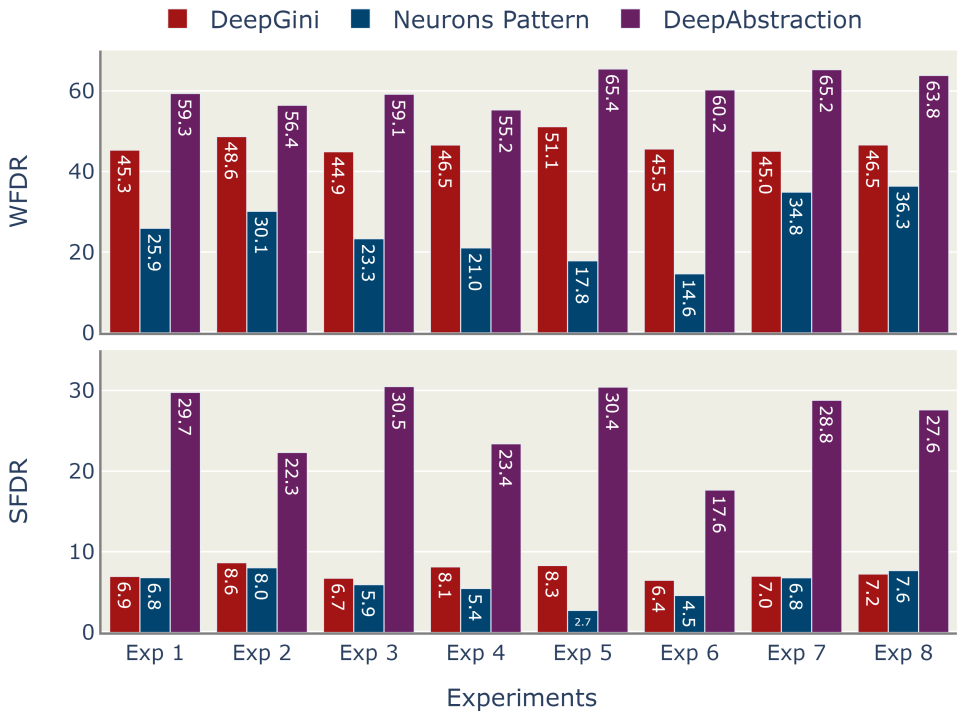


Figure 4.5: WFDR & SFDR evaluate different algorithms.

instance. High-severity instances have a deceptive similarity to the established pattern and thus have a high FD+ score, resulting in being at the bottom of the prioritization list.

Lastly, DeepAbstraction uses monitors to prioritize all rejected test instances at the beginning of the list. However, using the Gini score to prioritize these instances degrades the performance of DeepAbstraction as shown in Fig. 4.5. Table 4.5 confirms the good performance of DeepAbstraction by detecting many high and moderate-severity instances. Since the rate matters more than the ratio in the severity prioritization, the performance of DeepAbstraction cannot exceed 30%.

RQ 2 Answer :

The performance of all studied algorithms needs to be improved, with poor WFDR of less than 70% and SFDR values of no more than 30.5%. However, the DeepAbstraction algorithm shows significant improvement compared to the others in both measures.

4.6.3 *RQ3: Severity Distribution*

We investigate the significance of severity prioritization. In this regard, we evaluate the ratio of high-severity instances to the overall count of misclassified instances. Figure 4.6 provides a visual representation of this distribution, showing that high-severity instances are the most prevalent among the five experiments.

Moreover, the data shows that a considerable percentage, over 20%, of instances in every benchmark are of high severity. These figures underscore the prevalence of high-severity instances in our datasets, thus highlighting the depth of the problem. As a result, there is a pressing need to develop new algorithms that prioritize only highly severe instances at the top of the priority list.

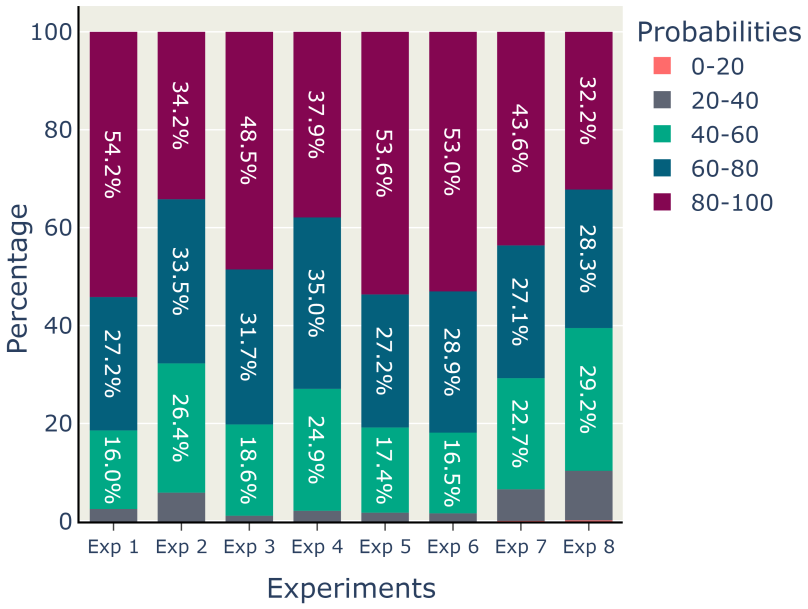


Figure 4.6: The distribution of different levels of severity.

RQ3 Answer :

Many instances in all benchmarks are highly severe, highlighting the need for new prioritization algorithms that specifically address the severity issue.

4.7 CONCLUSION

This chapter highlights the inefficacy of prevalent metrics like APFD, RAUC, and ATRC in evaluating prioritization algorithms. Specifically, APFD overlooks labeling costs, inadvertently inflating performance evaluations, while RAUC and ATRC depend heavily on FDRE rather than FDRO. Notably, current metrics disregard the misclassification ratio, an indicator of prioritization difficulty. To address this, we introduced the WFDR metric, which assigns weights based on prioritization difficulty. Our findings emphasize that a significant portion of datasets consists of highly severe test instances, underscoring the need for severity prioritization. This led to develop the SFDR metric, which is a top-weighted metric that evaluates the algorithm more heavily on the top of the list. Ultimately, our empirical tests validate the effectiveness of the WFDR and SFDR metrics and the poor performance of the studied algorithms.

5

DEEPABSTRACTION++: ENHANCING TEST PRIORITIZATION PERFORMANCE VIA COMBINED PARAMETERIZED BOXES

Main contributions of this chapter

- We comprehensively analyze the earlier version of DeepAbstraction, highlighting the weaknesses compromising its performance and reliability.
- We introduce the concept of *combined parameterized boxes* to leverage the collective verdicts of multiple monitors, enhancing the accuracy of our system's decisions.
- We establish a unique weighting system with a combination strategy that balances the decision-making process when conflicts arise among different verdicts of monitors, optimizing the fairness of the system's decisions.

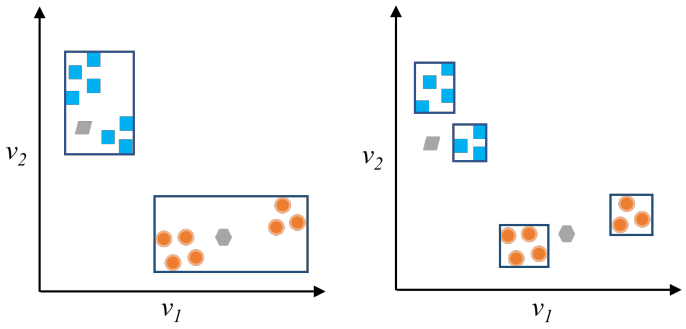
In this chapter, we start by taking a critical look at the earlier version of DeepAbstraction to highlight the limitations of the framework. We then introduce our approach in which we establish a unique weighting system and combination strategy. Finally, we aim to answer research questions about the effectiveness of verdict weights, how DeepAbstraction++ compares with other algorithms, and the stability of performance under varying parameter settings.

5.1 PROBLEM ANALYSIS

In this section, we explain the crucial role that clustering plays within the framework. We then discuss the challenge of tau selection and its consequential effects on the overall performance of the framework.

5.1.1 *Clustering*

Box-abstraction monitors are built based on the presumption that instances of the same class show similar patterns due to their greater contiguity within the feature space than instances of other classes. However, monitors may incorrectly validate the network's prediction for a new test input that closely mimics instances within a box, even though this instance originates from another class. Therefore, to alleviate this problem of false negatives, the k-means clustering algorithm is applied before box construction. After clustering, each cluster forms a small box rather than a large box for all clusters. Figure 5.1 illustrates how monitors in Fig. (a) falsely accept the predictions as a square and a circle for parallelogram and hexagon instances, respectively, i.e., novel classes not in the training dataset. After clustering in Fig. (b), the monitors correctly reject the predictions as they are outside all boxes.



(a) Without clustering (b) With clustering

Figure 5.1: Novel test instances before and after clustering [58].

5.1.2 *Tau Selection Issue*

DeepAbstraction controls the size of each box by a pre-specified parameter, namely the clustering parameter (τ), which has one of the following values: 0.4, 0.3, 0.2, 0.1, 0.05, and 0.01. The dynamic relationship between the value of τ and the box size is such that a decrease in τ value shrinks the box, whereas an increase expands it. The challenge is to select the best τ that optimally reduces the frequency of false negatives while enhancing the number of true positives. The choice of the ideal τ is deeply influenced by the inherent distribution of the dataset, which may vary across different classes. Therefore, DeepAbstraction lacks a definitive guideline for selecting the best tau across several benchmarks. For instance, DeepAbstraction suggests setting τ to 0.05 as a default value for models with a training accuracy of less than 98%, while a τ of 0.4 is for exceptionally accurate models. These values of τ are experimentally validated. However, these default

values of τ are empirical consensus rather than optimal values over all benchmarks, as shown in Fig. 5.2. For instance, when τ is 0.01 in the following experiments achieves better results than τ of 0.05: Exp.3 and Exp.6. Similarly, the performance of DeepAbstraction with τ of 0.4 in Exp.5 is less effective than with τ of 0.3.

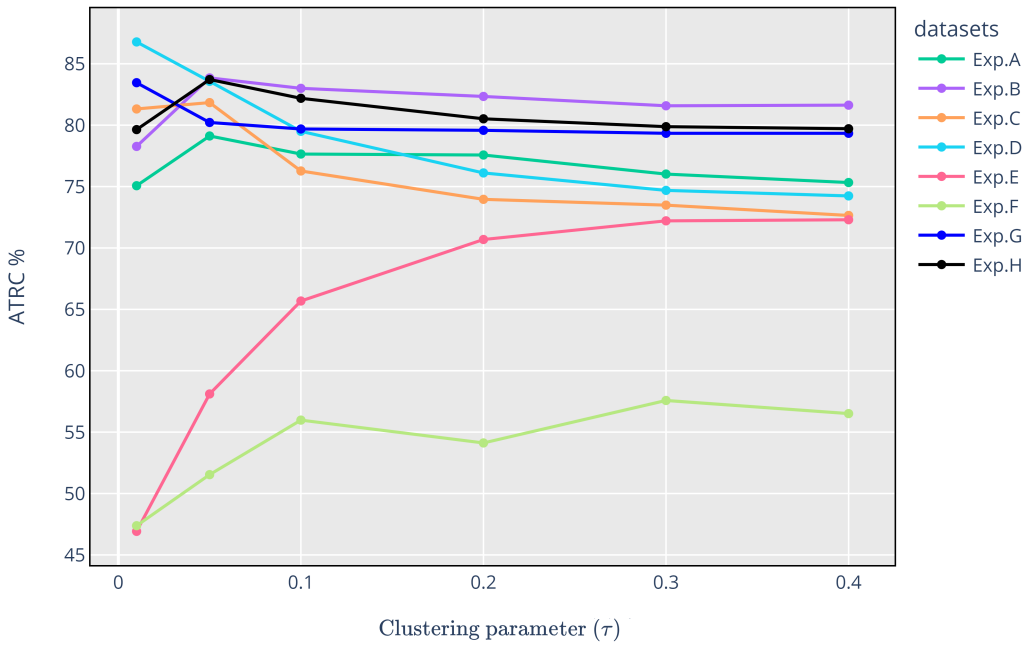


Figure 5.2: The effect of clustering parameter τ on performance[58].

5.2 APPROACH

In this section, we present the proposed solution, accompanied by its formal definitions. Then, we provide an illustrative example for the updated framework.

5.2.1 *Combined Parameterized Boxes*

Figure 5.2 illustrates that a predefined value of τ cannot effectively improve the performance. There is an observable discrepancy in the performance, i.e., small values of τ work in some cases better than large values, and vice versa. Therefore, we propose an inclusive approach that depends on all monitors' verdicts of different τ to accept or reject the neural network predictions. This integration represents so-called combined parameterized boxes, which collectively involve the predictive potential of every fixed-size box. With this approach, we effectively address the problem of τ selection.

The primary responsibility of monitors is to carefully reject any erroneous predictions. This task gains importance when disagreements arise among monitors of different τ. Amidst such conflict, if even a single verdict signals rejection, the final decision leans towards rejection. Our experimental evaluation further supports the effectiveness of this approach. In response to these findings, we develop a strategic approach to assign weights to the monitors' verdicts. This approach places a higher weight on rejection verdicts than other verdicts, while uncertainty verdicts carry more weight than acceptance ones.

It's crucial, however, to maintain a careful balance - the weight differences should not be so significant that the heavily weighted verdicts negate the lesser ones. For instance, overemphasis on rejection verdicts can cancel the contributions of other verdicts, thereby negatively impacting the prioritization performance in subsequent stages. To formalize this approach, we mathematically express the monitor's verdicts in the following order: acceptance [**a**], uncertainty [**u**], and rejection [**r**]:

$$\mathbf{a} = \gamma, \tag{5.1}$$
$$\mathbf{u} = \mathbf{a} + \beta, \tag{5.2}$$
$$\mathbf{r} = \mathbf{u} + 2 * \beta \tag{5.3}$$

where γ and β are arbitrary positive real numbers.

We start to randomly select the values of γ and β. Then we compute the weights of the verdicts according to the above equations. We can also observe that the acceptance weight can be any positive real number except zero since zero denotes no contribution. Furthermore, the uncertainty weight is greater than the acceptance weight with β. Moreover, the rejection weight is larger than the uncertainty weight by $2 * \beta$. Lastly, if we substitute eq.(5.2) in eq. (5.3), we infer that the rejection weight is larger than the acceptance weight by $3 * \beta$. In the experimental evaluation section, we will see how different values of β should not affect the performance stability of the combined monitors. In other words, the performance stability is independent of the β selection value.

5.2.2 *Combination Strategy*

Numerous strategies exist to merge the weighted verdicts of different monitors and yield a final, cumulative verdict. We highlight a few of these approaches below, with a more comprehensive evaluation of each to follow in the experimental evaluation section:

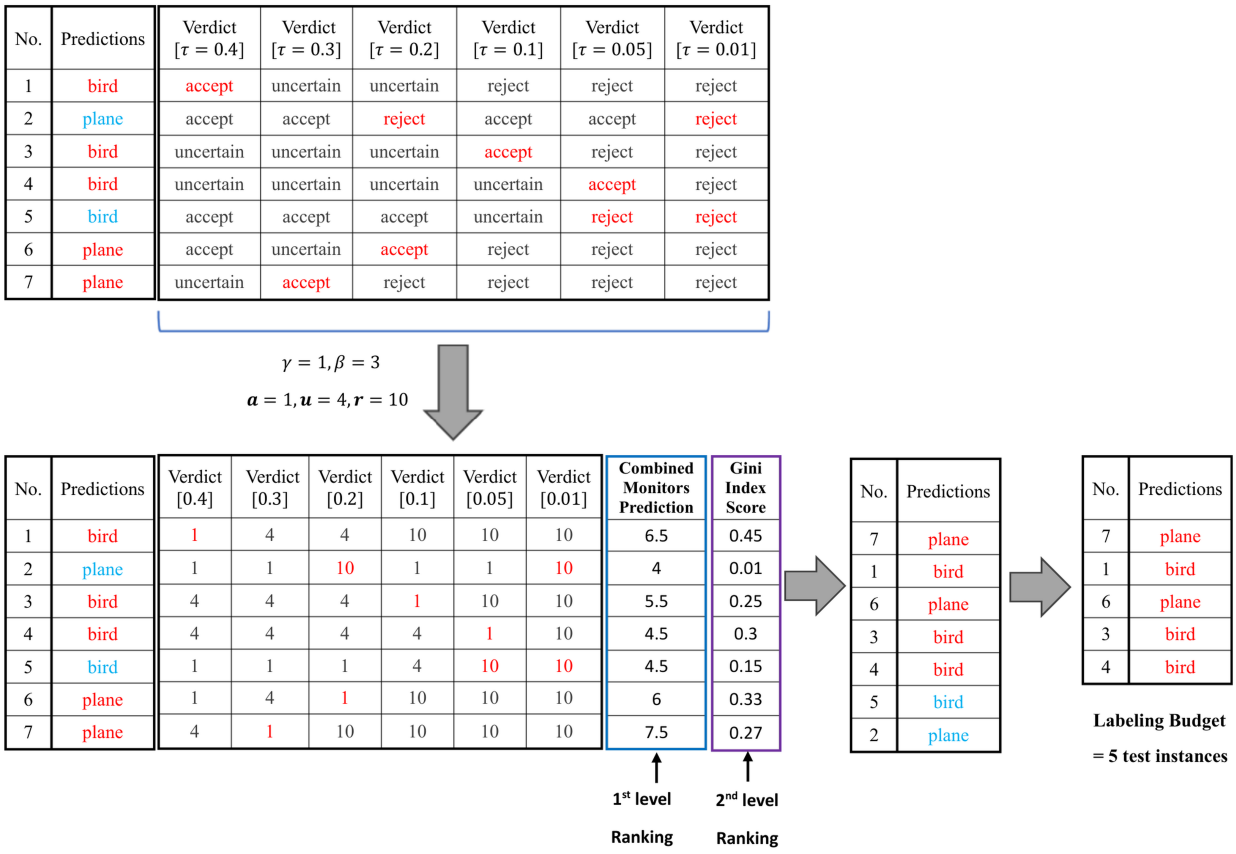
- *Mean:* This strategy involves adding all weighted verdicts and dividing the total by the number of verdicts, in this case, six—corresponding to the τ values of 0.4, 0.3, 0.2, 0.1, 0.05, and 0.01.
- *Max:* This strategy selects the largest weighted verdict among the six. If a rejection is present, it automatically becomes the final verdict, followed by uncertainty or acceptance verdicts, as applicable.
- *Product:* As the name suggests, this strategy takes the product of all weighted verdicts as the final verdict.
- *Mode:* This fundamentally operates as a voting strategy, where the final verdict is the weighted verdict appearing most frequently among the others.

By evaluating these strategies, we aim to provide insight into their efficacy and relevance in the context of our monitoring system.

5.2.3 *Illustrative Example*

Imagine we task a neural network with a binary classification scenario where it should differentiate between plane and bird. As depicted in Fig. 5.3, the predictions highlighted in

red represent misclassifications and should be prioritized. Conversely, the ones highlighted in blue indicate the correct classification. We can observe that the upper part of Fig. 5.3 shows DeepAbstraction version 1, which consists of 6 verdicts according to the τ value. In this version, the user should select *only one* verdict depending on the τ value determined by the model's training accuracy. However, there are some cases where training accuracy does not sufficiently capture the model's learning capability, e.g., overfitting. Ultimately, it's crucial to consider verdicts of false negatives and false positives, which are marked in red, to assess the predictive performance of the monitors.



No.	Predictions	Verdict [τ = 0.4]	Verdict [τ = 0.3]	Verdict [τ = 0.2]	Verdict [τ = 0.1]	Verdict [τ = 0.05]	Verdict [τ = 0.01]
1	bird	accept	uncertain	uncertain	reject	reject	reject
2	plane	accept	accept	reject	accept	accept	reject
3	bird	uncertain	uncertain	uncertain	accept	reject	reject
4	bird	uncertain	uncertain	uncertain	uncertain	accept	reject
5	bird	accept	accept	accept	uncertain	reject	reject
6	plane	accept	uncertain	accept	reject	reject	reject
7	plane	uncertain	accept	reject	reject	reject	reject

No.	Predictions	Verdict [0.4]	Verdict [0.3]	Verdict [0.2]	Verdict [0.1]	Verdict [0.05]	Verdict [0.01]	Combined Monitors Prediction	Gini Index Score
1	bird	1	4	4	10	10	10	6.5	0.45
2	plane	1	1	10	1	1	10	4	0.01
3	bird	4	4	4	1	10	10	5.5	0.25
4	bird	4	4	4	4	1	10	4.5	0.3
5	bird	1	1	1	4	10	10	4.5	0.15
6	plane	1	4	1	10	10	10	6	0.33
7	plane	4	1	10	10	10	10	7.5	0.27

No.	Predictions
7	plane
1	bird
6	plane
3	bird
4	bird
5	bird
2	plane

No.	Predictions
7	plane
1	bird
6	plane
3	bird
4	bird

Figure 5.3: Transition from DeepAbstraction to DeepAbstraction++.

Our process begins by assigning weights to the acceptance, uncertainty, and rejection verdicts, according to eq. (5.1)-(5.3), yielding respective values of 1, 4, and 10. Then, we incorporate the verdicts of different monitors using the *mean* combination strategy to ascertain the final verdict. Following this, we prioritize the test instances based on the final verdict. However, in cases where two test instances possess an equal combined verdict score, we turn to the GI score for prioritization. For example, the fourth and fifth test instances have a combined verdict of 4.5, prompting the algorithm to prioritize the fourth over the fifth based on their GI scores. Finally, after the prioritization completion, we label the first n test instances. Here, n represents the predetermined labeling budget, setting the threshold for the number of instances to be labeled.

5.3 EXPERIMENTAL SETUP

We conduct the experiments on a system equipped with an Nvidia K80 GPU and 12 GB of RAM, with PyTorch v1.9.0 as the underlying framework. Table 5.1 summarizes the principal experiments. The configurations used for the primary setup are as follows:

- **Datasets**: MNIST [15], Fashion-MNIST [85], CIFAR10 [34], SVHN [46].
- **Pretrained Model**: ResNet18 [27], GoogLeNet [74], ResNet34 [27], ResNet50 [27], ResNet101 [27], ResNet152 [27], and EfficientNet-B0 [75].
- **Prioritization Algorithms**: DeepGini[22], DeepAbstraction[58], and DeepAbstraction++.
- **Evaluation Metrics**: We use the WFDR metric to evaluate prioritization algorithms, according to [57]. This metric outperforms other metrics in effectively assessing the quality of prioritization algorithms. Unlike other metrics, the WFDR metric involves the prioritization difficulty which highly depends on the dataset size and the labeling budget.
- **Research Questions**:
  - ❶ **(Weights Effectiveness)**: How effective are the verdict weights proposed in eq. (5.1)-(5.3)?
  - ❷ **(Algorithms Effectiveness)**: How effective is DeepAbstraction++ compared to the state-of-the-art (SOTA) algorithms?
  - ❸ **(Performance Stability)**: How does tuning the γ and β parameters influence the performance of DeepAbstraction++?
  - ❹ **(Combination Strategy Selection)**: Which combination strategy provides better performance in terms of algorithm effectiveness?

Table 5.1: Details of the datasets and pretrained models.

Exp ID	Dataset	Training Dataset	Test Dataset	Pretained Model	Training Acc. (%)	Test Acc. (%)
Exp 0	CIFAR-10	50,000	10,000	Efficient-B0	94.95	92.86
Exp 1	CIFAR-10	50,000	10,000	ResNet101	88.83	86.97
Exp 2	F-MNIST	60000	10000	Efficient-B0	94.94	94.17
Exp 3	F-MNIST	60,000	10,000	ResNet50	93.11	91.12
Exp 4	MNIST	60,000	10,000	ResNet18	99.36	99.16
Exp 5	MNIST	60,000	10,000	ResNet34	99.29	98.84
Exp 6	SVHN	73,257	26,032	GoogLeNet	95.51	95.07
Exp 7	SVHN	73,257	26,032	ResNet152	94.63	94.10

5.4 EXPERIMENTAL EVALUATION

This section addresses the research questions outlined in the previous section. First, we assess the effectiveness of the verdict weights proposed in eq. (5.1)-(5.3). Then, we evaluate the efficacy of DeepAbstraction++ algorithm by contrasting its performance with the SOTA algorithms. Third, we explore the performance stability of DeepAbstraction++ by examining the impacts of tuning the parameters γ and β. Finally, we will determine the best combination strategy.

5.4.1 *Weights Effectiveness*

We perform eight experiments as detailed in Table 5.1 where the combination strategy is the mean, and γ and β are 1 and 3, respectively. In our initial experiment, we aim to confirm the necessity of assigning greater importance to the weight of a rejection verdict compared to other verdicts. Specifically, we study how a single rejection verdict can influence the final combined verdict compared to the other five verdicts.

Our findings suggest that when only one monitor issues a rejection verdict, this results in a final combined verdict of rejection in 30% of misclassified instances, as indicated in Exp.0 of Fig. 5.4. The last ratio is significantly greater in Exp.1, 3, and 7, standing at 47%, 48%, and 48%, respectively. However, we found that in scenarios where the neural network exhibits high levels of accuracy, a single *rejection* verdict is insufficient to refuse the network prediction, as evidenced by the results of Exp.4 and 5 in Fig. 5.4. When the number of rejection verdicts increases to three or six, we can further confirm this finding. For example, in Exp.0, we noticed that of the instances resulting in a final combined verdict of true rejection, 39% had three rejection verdicts, and 73% had six rejection verdicts. We consistently observe this trend across all the conducted experiments. It strongly underlines the pivotal role that rejection verdicts play in determining the final decision over other verdicts.

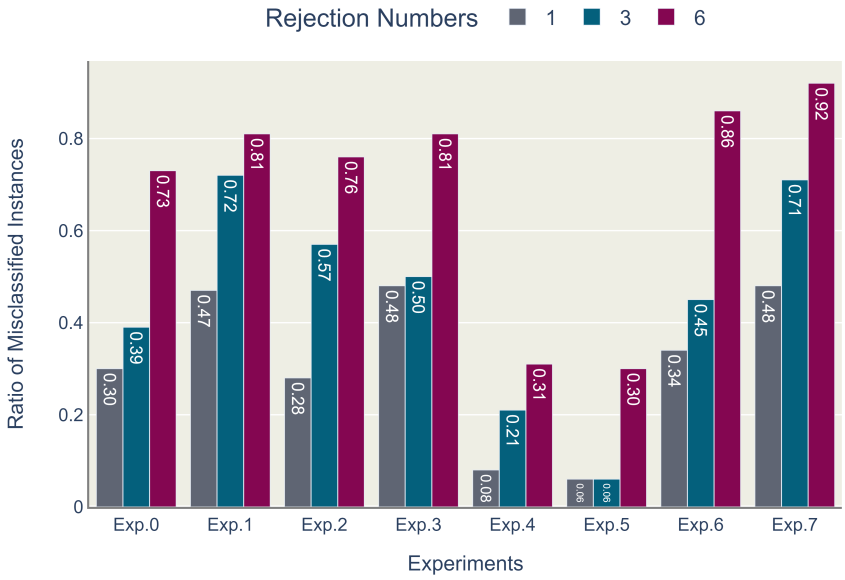


Figure 5.4: The impact of the number of rejection verdicts on the final combined verdict.

In the second experiment, we contrast the impact of the rejection verdict compared to the other two types: uncertainty and acceptance. The aim is to investigate the influence each verdict type has on the true rejection of the final decision. More specifically, our comparison involves only those instances where the verdicts are consistent across all six monitors. Then, we compute the number of instances in which the six rejection verdicts led to the true rejection of the final verdict. After that, we compare this with the total number of six rejection-verdict instances to find the ratio. This procedure is repeated with instances of six uncertainty verdicts and six acceptance verdicts.

Our findings, as depicted in Fig. 5.5, highlight the considerable influence of the rejection verdicts on the final combined verdict. In six out of eight experiments, this verdict type significantly outperformed the others, with the highest contribution ratio reaching 91% and a median ratio of 78%. On the contrary, instances of uncertainty verdicts exhibit a moderate influence, with a maximum contribution ratio of 42% towards the true rejection of the final verdict. Instances of acceptance verdicts, however, demonstrate minimal impact on rejecting the final verdict.

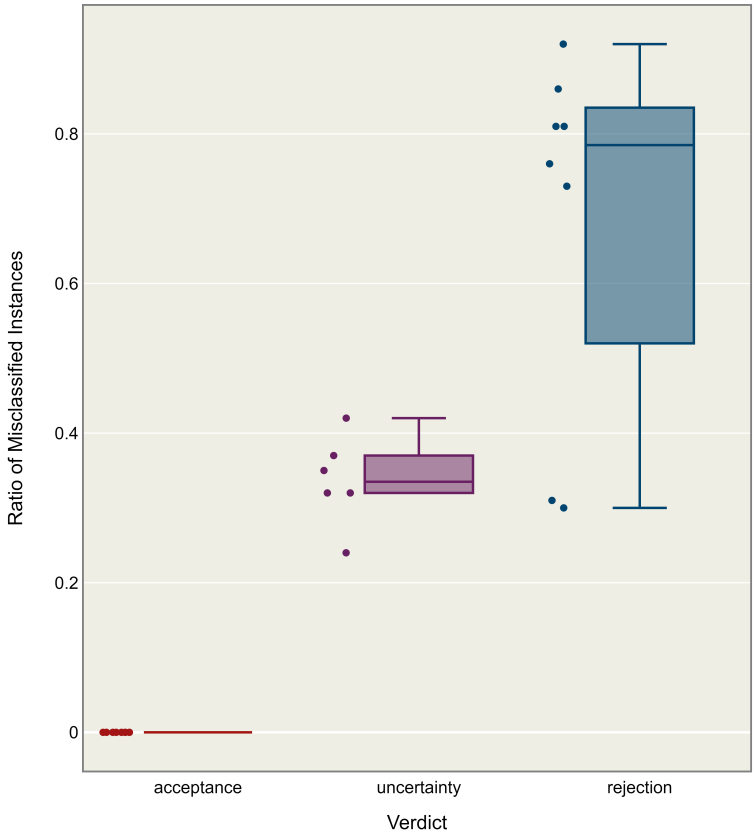


Figure 5.5: The impact of various verdict types on the final combined verdict.

RQ 1 Answer :

Our study reveals that the observed impact of the verdicts on the final verdict aligns with the proposed weights of the verdicts in the eq.(5.1)-(5.3).

5.4.2 *Algorithms Effectiveness*

Table 5.2 presents a comparative study on the effectiveness of the DeepGini, DeepAbstraction, and DeepAbstraction++ algorithms, evaluated using the WFDR metric across eight experiments. In each experiment, the combination strategy is the mean, and γ and β are 1 and 3. Table 5.2 reveals that DeepAbstraction++ consistently outperforms both DeepGini and DeepAbstraction in all experiments, as shown by the positive deltas. The improvements offered by DeepAbstraction++ over DeepAbstraction range from a minimum of +2.38% (in Exp.4) to a maximum of +7.71% (in Exp.6). This demonstrates that the additional optimizations in the DeepAbstraction algorithm are effective.

Table 5.2: Effectiveness of DeepAbstraction++ and other algorithms (WFDR).

Experiment	DeepGini (%)	DeepAbstraction (%)	DeepAbstraction++ (%)	Δ
Exp.0	45.26	58.75	64.26	↑+5.51
Exp.1	48.62	56.39	59.90	↑+3.51
Exp.2	44.86	59.13	63.79	↑+4.66
Exp.3	46.53	53.19	59.21	↑+6.02
Exp.4	51.08	65.38	67.76	↑+2.38
Exp.5	45.54	60.20	62.79	↑+2.59
Exp.6	45.00	67.59	75.30	↑+7.71
Exp.7	46.55	63.79	69.70	↑+5.91

RQ 2 Answer :

DeepAbstraction++ demonstrates considerably higher effectiveness than other algorithms. Therefore, the new additions to the framework greatly enhance the performance.

5.4.3 *Performance Stability*

As demonstrated in Fig. 5.6, the DeepAbstraction++ model exhibits remarkable stability in performance. Regardless of the β value, which ranges from 1 to 5000, the performance remains constant for each experiment. This consistent performance across a broad spectrum of β values indicates a high level of stability in DeepAbstraction++ performance. Similarly, when the parameter β is held constant at a value such as 1, and γ varies within a range from 1 to 5000, we consistently observe the stable performance of the DeepAbstraction++ model across all γ values for every experiment. For the sake of brevity, the corresponding graph is omitted as it is highly similar to Fig. 5.6.

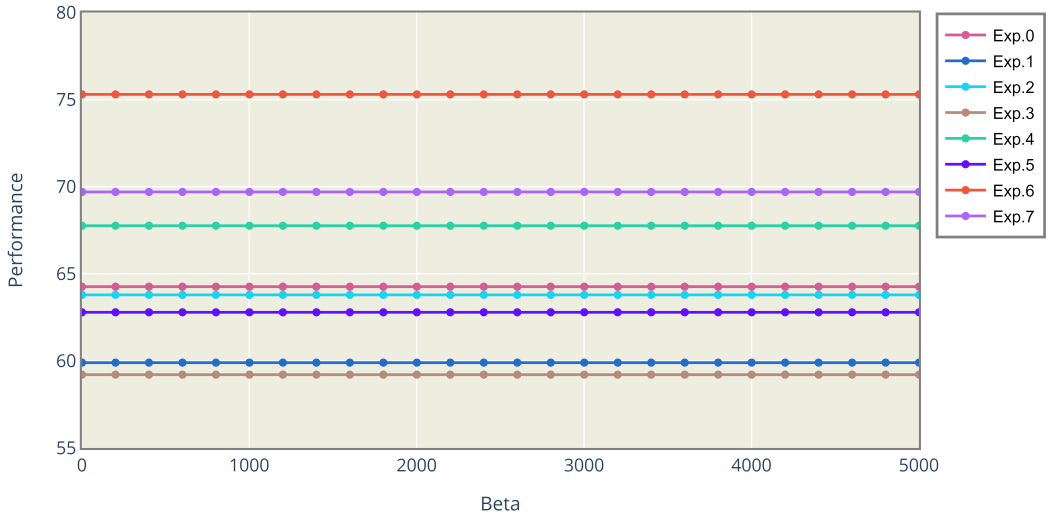


Figure 5.6: The Impact of β on the performance stability when $\gamma = 1$.

RQ 3 Answer :

DeepAbstraction++ consistently maintains stability, unaffected by the values of γ and β, indicating that γ and β do not impact the performance.

5.4.4 *Combination Strategy Selection*

Table 5.3 compares the effectiveness of different strategies where γ and β are 1 and 3. The evaluation is based on the WFDR percentage, incorporating four strategies: mean, product, mode, and max. The mean strategy generally outperforms the other strategies in all the experiments, with the highest mean value seen in Exp.6 at 75.30%. However, there are exceptions to this pattern. For instance, Exp.4 presents a noteworthy difference between the mean (67.76%) and the max (57.04%). Moreover, in Exp.5, the mean and product values are identical at 62.79%. Other combination strategies tend to perform badly because they need to include all monitors in their final combined verdict. For instance, the max strategy only chooses the maximum verdict over the six verdicts. Also, the mode strategy is biased towards the majority verdicts rather than incorporating all. Additionally, the product strategy prioritizes the rejection verdict above other verdicts when determining the final combined verdict. However, product strategy works better when all verdicts are rejections according to Fig. 5.4. On the other hand, the mean strategy manages to incorporate all monitors when determining the final verdict.

RQ 4 Answer :

The results suggest a prevailing superiority of the mean strategy in merging the verdicts from multiple monitors over other strategies.

Table 5.3: Comparative analysis of strategies based on WFDR (%).

Experiment	mean (%)	Product (%)	mode (%)	max (%)
Exp.0	**64.26**	58.76	59.17	60.30
Exp.1	**59.90**	59.43	55.45	59.95
Exp.2	**63.79**	63.45	58.09	60.68
Exp.3	**59.21**	58.42	52.51	56.70
Exp.4	**67.76**	65.38	64.19	57.04
Exp.5	**62.79**	**62.79**	61.07	52.44
Exp.6	**75.30**	73.41	67.19	65.15
Exp.7	**69.70**	68.98	62.02	66.41

5.5 CONCLUSION

This chapter explores the issue of tau selection in the DeepAbstraction framework, a factor that affects the box size as well as the framework's stability and performance. We present a new method called *combined parameterized boxes*. This method takes into account judgments from monitors using various tau values to evaluate network predictions. We give these judgments specific weights to avoid undue influence from any single type of verdict, aiming for a more balanced decision-making process. We also suggest several strategies like mean, max, product, and mode to integrate these weighted judgments into a final decision. The method offers a considerable improvement in the performance of the DeepAbstraction framework.

Part III

STATE OF THE ART & RELATED WORK

6

RELATED WORK

In this chapter, we focus on two key concepts: *test prioritization algorithms* and *runtime monitoring*.

We start this chapter with test prioritization, highlighting the recent work published in this area. We explore various techniques, e.g., entropy, DeepGini, PRIMA, and TestRank, among others. These methods collectively aim to enhance the efficiency of DNN testing by prioritizing inputs that are most likely to reveal defects or anomalies in the model. In this chapter, we also present runtime monitoring, a concept that our work heavily relies on to develop our test prioritization framework. More importantly, the correlation between test prioritization algorithms and runtime monitoring in this chapter lies in their shared objective of enhancing the robustness and reliability of DNNs.

6.1 TEST PRIORITIZATION ALGORITHMS

In this section, we list several state-of-the-art algorithms in the area of test prioritization.

6.1.1 *Input prioritization*

The paper [6] emphasizes the significance of testing neural networks, especially in safety-critical systems. The authors propose a novel method to prioritize input data to reduce the labeling cost. The authors also introduce the use of sentiment measures, derived from the computations performed by the model. These measures help identify inputs that might reveal the model's weaknesses. The primary methods discussed for test prioritization include:

6.1.1.1 *Entropy*

This measure [69] uses the output of the softmax function that represents the categorical probability distribution of the classes. To determine how *uncertain* the model is about its prediction for a given input, the concept of *entropy* is used. Entropy measures the randomness or uncertainty in these probabilities. A high entropy value means the model is uncertain about its prediction, while a low value indicates confidence. In the context of test prioritization, inputs that the model is uncertain about, i.e., high entropy, are given higher priority for testing since they might reveal potential model weaknesses. The entropy-based score is formulated in eq.(2.5).

6.1.1.2 *Monte-Carlo Dropout*

In machine learning, the model uncertainty can arise due to limited training data or inherent noise in the data –known as aleatoric uncertainty [31]. Bayesian Neural Networks (BNNs) [65] provide a way to estimate this uncertainty.

Monte-Carlo Dropout [72], originally a regularization technique for neural networks, can also function as a Bayesian approximation [24]. Unlike its standard use where Dropout is disabled during testing, Monte-Carlo Dropout keeps it active. This change allows each test run to sample from a network with a randomly selected subset of active neurons, effectively creating a probability distribution over the network's predictions. This process imitates Bayesian methods, where uncertainty is represented probabilistically. By averaging the results of multiple forward passes with active Dropout, the network's predictive uncertainty is estimated. Thus, Monte-Carlo Dropout not only prevents overfitting but also provides a Bayesian perspective on the network's predictions, highlighting the confidence level in its outputs.

When a model is uncertain about its prediction for a particular input, that input becomes a potential candidate for further testing. The Monte-Carlo Dropout approach offers an approximation way to estimate this uncertainty, making it a valuable tool for prioritizing test inputs.

6.1.1.3 *Distance-based Surprise Adequacy*

The DSA method [32] is rooted in the concept of measuring the *surprising* or novelty of a test input. The idea is to assess how different or surprising a new input is compared to

known inputs based on the activation traces of the neural network. An activation trace is essentially the output of neurons in the network when processing an input. To compute the surprise of a new input, DSA utilizes a distance function. It first identifies the closest known input with the same predicted class to the new input. Then, it finds another known input that is closest to the first one but has a different predicted class. The distances between the activation traces of these inputs are then used to compute the DSA value for the new input. A higher DSA value indicates that the new input is more surprising to the model, suggesting that the model might not be well-prepared for such an input. As a result, inputs with high DSA values are prioritized for testing.

In essence, the DSA method prioritizes test inputs based on how different their activation patterns are compared to known inputs, with the belief that more *surprising* inputs are likely to reveal potential weaknesses in the model.

To conclude, Byun et al.[6] were among the first to introduce sentiment measures such as confidence, surprise, and uncertainty to address test prioritization. Their methodology is critically limited by its reliance on the model's output—measures that can be misleading due to overconfidence in predictions. This reliance leads to an incorrect estimation of uncertainty and potential misclassification.[23].

6.1.2 *DeepGini*

DeepGini [22] is a testing methodology for DNNs designed to enhance robustness. Unlike conventional techniques that focus on maximizing the coverage of neurons or pathways [54], DeepGini uses the Gini Index (GI) [59], a metric for assessing inequality in distributions. For more details about GI, we refer the reader to Section 2.8.2. DeepGini applies the GI to the outputs of neurons in the penultimate layer. The approach ranks test instances according to the GI which reflects the impurity or uncertainty in the DNN's outputs. Specifically, when a DNN showcases similar probabilities across all classes, it indicates a higher likelihood of misclassification due to uncertainty, and vice versa. By prioritizing such tests, DeepGini addresses the most vulnerable areas of the DNN, thereby enhancing the overall robustness.

However, DeepGini proposes using the Gini Index to identify error-prone tests, it falls short by depending solely on model outputs to assess test case uncertainty. This reliance does not take into account the full complexity of DNN behavior and often misses high-confidence misclassifications, particularly with near-centroid instances[23].

6.1.3 *PRIMA*

The paper [82] introduces **PR**ioritizing test inputs via **I**ntelligent **M**utation **A**nalysis (PRIMA), a novel test input prioritization approach. The primary goal of PRIMA is to efficiently label test inputs that reveal bugs, thereby enhancing the efficiency of the DNNs testing. PRIMA utilizes a two-fold strategy for test prioritization:

1. **Mutation Analysis**: PRIMA's approach to test prioritization is deeply rooted in mutation analysis [81], which is based on two foundational insights:
    - *Model Mutation*: By introducing slight modifications to the DNN model, PRIMA produces various mutated versions. When a test input yields different prediction results between the original model and one of its mutated counterparts, the test

input is said to "kill" the mutated model. This behavior indicates that the test input effectively tests the part of the model that was altered. The underlying rationale is that if a test input can highlight differences between the original and a slightly changed model, it's more likely to reveal potential bugs in the DNN. PRIMA uses various mutation rules, such as:

a. *Neuron Activation Inverse (NAI)*: Inverts the activation state of a neuron.

b. *Neuron Effect Block (NEB)*: Blocks the effect of a neuron on subsequent layers.

c. *Gauss Fuzzing (GF)*: Adds noise to neuron weights.

d. *Weights Shuffling (WS)*: Shuffles the weights of a neuron.

- *Input Mutation*: PRIMA also slightly alters test inputs, creating mutated versions of test instances. If a significant number of these mutated versions produce different prediction results compared to the original test input on the unaltered model, this suggests a notable finding. It indicates that the original test input is sensitive and adept at capturing DNN anomalies. This is because the original test input's information is being effectively utilized by the model, and even slight changes to the input can lead to different outcomes.

2. **Learning-to-Rank**: is a specialized form of supervised machine learning, that plays a pivotal role in test prioritization. The process begins with the extraction of features from mutation results, capturing the prediction differences between the original and mutated models or inputs. Leveraging these features, PRIMA uses the XGBoost ranking algorithm [12] to construct a model that can rank test inputs depending on their potential to reveal defects in DNNs. Once this model is established, it assigns scores to each test input. Those with higher scores are perceived as more likely to uncover bugs, and as a result, they are prioritized for testing. Through this intelligent ranking system, PRIMA ensures that the most critical tests, which are most likely to highlight vulnerabilities in the DNN, are addressed first.

However, PRIMA introduces a mutation-based strategy that, while innovative, struggles with scalability due to the high storage requirements for multiple models and test inputs. It is also further compromised by the unpredictability associated with random mutations.

6.1.4 *TestRank*

The paper [39] presents TestRank, a novel test input prioritization framework. TestRank seamlessly integrates both intrinsic and contextual attributes of the test data:

1. **Intrinsic Attributes Extraction**: TestRank extracts the intrinsic attributes directly from the target deep learning model. For each input in the pool of unlabeled inputs, TestRank collects the output logits, the vectors present before the softmax layer. This process captures the inherent characteristics of the test data, providing a foundational layer of information for prioritization.

2. **Contextual Attributes Extraction**: Initially, TestRank first maps the original data space into a more compact feature space, preserving a strong local continuity property [2]. A feature extractor carries out this transformation. Then, TestRank constructs a similarity graph, specifically a k-Nearest Neighbor Graph [33]. This graph draws

from the feature vectors and their associated classification correctness, including both the unlabeled test pool data and labeled data like training sets or previously labeled test samples. Using a graph-based representation learning technique, TestRank extracts contextual attributes for each unlabeled instance. The goal here is to use the classification correctness of similar labeled samples as a guide for prioritizing unlabeled ones.

3. **Combining Intrinsic and Contextual Attributes**: TestRank combines the intrinsic and contextual attributes using a Multi-Layer Perceptron (MLP). This MLP predicts the potential of unlabeled test instances to reveal failures. After making these predictions, TestRank ranks the instances, prioritizing those with the highest potential to reveal failures.

In summary, TestRank solves the deficiencies of its predecessors by integrating a broader range of network attributes. However, its effectiveness is hampered by the necessity of pre-labeled test instances as a prerequisite which in turn reduces the applicability in constrained datasets or budgets. In addition, TestRank shows unstable performance across different models of the same dataset.

6.1.5 *Neurons Pattern*

The paper [87] presents an innovative test prioritization approach. It introduces the concept of a *familiarity score*, denoted as **FR+**. This score quantifies the familiarity of a given input based on neuron activation patterns observed during training. The primary idea is that inputs with lower familiarity scores are more *unfamiliar* or *novel* to the DNN. Thus, they are given higher priority for testing as they are more likely to expose potential issues in the model. This approach aims to enhance the efficiency of identifying problematic classifications in DNNs.

To establish this prioritization, the paper first constructs patterns based on the training set. These patterns reflect the active and inactive neurons for each class in the DNN. Once these patterns are established, the test cases are then prioritized based on their familiarity degrees to these patterns. A higher familiarity degree indicates that the test case is more similar to the training data, and thus, it is given a lower priority. Conversely, test cases that deviate more from the established patterns, indicating potential unfamiliarity or novelty, are given higher priority. This method operates on the premise that such unfamiliar test cases are more likely to reveal errors in the DNN.

6.1.6 *ActGraph*

ActGraph [10] introduces a distinctive approach to test case prioritization by emphasizing the spatial relationships of neurons in DNNs. This relationship is captured using an activation graph which is originally a directed weighted graph [71]. The graph integrates the structure of the DNN and the multi-layer activation values, providing a holistic representation of the test case's features.

The initial step involves feeding test cases into a trained DNN, with each layer producing activation values. These activations shed light on how different neurons in the network respond to the input, providing insights into the DNN's behavior for that specific test case. Following this, ActGraph constructs activation graphs based on these activations. This

graph is a directed weighted structure where neurons are nodes, and the model's weights act as connections between them. From this graph, an adjacency matrix and node features are extracted, representing the relationships and characteristics of neurons, respectively.

The process refines these features by extracting a central node feature from the activation graph. This extraction offers a summarized representation of the test case's behavior in the DNN. Using this feature, ActGraph employs a Learning-to-Rank (L2R) model [12] to rank test cases based on their potential to uncover vulnerabilities in the DNN. This ensures that test cases with a higher likelihood of exposing model flaws are prioritized, enhancing the efficiency of DNN testing.

6.1.7 *CertPri*

CertPri [92] is a novel method for test input prioritization in DNNs, focusing on the movement cost in DNNs feature space. This approach quantifies the distance a test input needs to move to be correctly classified, known as *movement cost*. CertPri evaluates this cost to prioritize test inputs, particularly those that are misclassified or reveal bugs, aiding in improving the model performance. Furthermore, CertPri provides a formal robustness guarantee for this prioritization process, employing the Lipschitz continuity assumption [53]. This guarantee enhances the reliability and effectiveness of the method across various types of tasks and data models.

CertPri's methodology is fundamentally based on two key concepts: feature purity [22] and the use of inverse perturbation [91]. Feature purity relates to the stability of test inputs in the feature space, where high purity indicates stability and necessitates a higher movement cost for further change. This stability is crucial during forward propagation [11] as inputs move towards their class centers. Inverse perturbation, conversely, is used to analyze the movement cost for both correctly and incorrectly predicted test inputs. It improves the feature purity of incorrectly predicted inputs, enabling them to move directionally in forward propagation with lower movement costs. In contrast, correctly predicted inputs with higher feature purity require more effort to reach their class centers. This differential treatment based on feature purity and inverse perturbation underlines CertPri's effectiveness in handling diverse inputs and enhancing DNN testing and quality assurance.

6.1.8 *DeepHyperion-CS*

DeepHyperion-CS [96] introduces a methodology that explores the feature space of deep learning systems to enhance testing. This tool enhances its predecessor, DeepHyperion [95], by emphasizing inputs that significantly contribute to the exploration of the feature space during previous search iterations. The goal of this exploration is to understand the specific features or characteristics of test inputs that influence the behavior of the DL system. Moreover, DeepHyperion-CS is designed to explore a DL system's feature space extensively, seeking inputs with diverse characteristics that cause the system to deviate from expected behavior. Using Illumination Search algorithm [45], DeepHyperion-CS fills the feature map with inputs that either expose or are close to exposing misbehaviors in the DL system.

DeepHyperion-CS prioritizes test inputs based on their contribution to feature space exploration. Inputs that have significantly contributed to previous explorations of the

feature space receive a higher priority. This prioritization is achieved by assigning a *CS score* to each input. Inputs with a history of contributing more to the exploration of the feature space receive a higher CS score, making them more likely to be selected in subsequent iterations. If an individual input repeatedly shows a lack of contribution to the exploration, the system progressively reduces the CS score of that input, thereby lowering the priority during selection. This dynamic adjustment ensures efficiency in the exploration process by focusing on inputs more likely to expose new areas of the feature space and potential misbehaviors of the DL system.

6.1.9 *Activation Frequency*

The paper [88] introduces a method for prioritizing test cases in DNNs. Researchers analyze the activation pattern of neurons within the DNN. Specific inputs activate neurons, and the pattern of these activations reveals insights into the network processing. The method calculates the activation frequency of neurons from training sets. This frequency shows how often a neuron activates with different inputs. By understanding frequent neuron activations, the method identifies inputs likely to trigger unusual or unexpected activations.

The method assigns high priorities to test cases more likely to be misclassified. This prioritization relies on the analysis of neuron activation patterns. Test cases that trigger unusual neuron activations receive higher priority. These test cases uncover misclassifications within the tested DNN and trigger misclassifications.

6.2 RUN-TIME MONITORING

In this section, we explain the three runtime monitoring works, we relied on in our work.

6.2.1 *Outside the Box*

The paper [28] introduces a novel framework for monitoring neural networks, addressing the challenge of novelty detection [48]. The framework's main goal is to identify previously unseen inputs, known as novel behaviors, by observing hidden layers within the neural network. Unlike traditional methods that rely on output confidence, this approach employs a common abstraction called *boxes* to recognize patterns in the monitored layers. The ability to detect novel inputs is crucial for understanding neural network decisions and is particularly vital in safety-critical applications like autonomous vehicles.

The mechanism of monitoring involves a three-phase process. First, the system collects outputs at a specific layer for inputs of a particular class. Then, it clusters these vectors based on their region. lastly, it constructs a box abstraction for each combination of class and cluster, resulting in a list of abstractions for each class. These boxes, or intervals, overapproximate the set of known neuron valuations. This overapproximation provides a simple and efficient way to represent and manipulate the data. The box abstraction is central to the framework's novelty detection performance, allowing for runtime monitoring without significant overhead.

At runtime, the monitoring procedure works in parallel with the neural network. It observes the output vector at a specific layer and compares it with the previously constructed

abstractions. If the observed vector fits within one of the abstractions, the monitor accepts the prediction, recognizing it as a typical behavior. If not, it rejects the input, signaling an atypical or novel behavior. This approach offers a more nuanced and efficient way to detect novel inputs, enhancing the reliability and interpretability of neural network classifiers. This approach represents a significant step forward in the field. It provides a flexible solution that can be tailored to achieve a balance between false warnings and undetected novelties.

6.2.2 *Customizable Runtime Monitoring*

The paper [83] presents a novel approach to runtime verification of classification systems through data abstraction. The focus is on detecting inputs that do not belong to the classes for which ML engineers have prepared the neural networks. The main goal of this monitoring is to enhance the trustworthiness and safety of AI-based systems, particularly in safety-critical applications. The paper introduces the concept of box abstraction with a resolution. This concept allows for a more precise representation of the neural network's behavior. It also provides insights into the relationship between clustering parameters and monitor performance.

The mechanism of monitoring is built upon geometrical shape abstraction, specifically using box abstraction. The paper extends previous work [28] by utilizing both correct and incorrect behaviors of a classification system to build box abstractions. These boxes represent sets of contiguous n-dimensional vectors constrained by real intervals. The approach introduces uncertainty verdicts, allowing the identification of suspicious regions when abstractions of good and bad references overlap. By controlling the space of a box, the notion of clustering coverage is introduced, serving as a quantitative metric that indicates the quality of the abstraction. This enables the study of the effect of different clustering parameters on the constructed boxes and the estimation of an interval of suboptimal parameters.

The paper also emphasizes the importance of clustering in constructing the monitor. Clustering algorithms, such as k-means [1], are applied to determine a partition of the set based on similarity measures. This leads to a more precise abstraction of the sets as the union of the boxes computed for each cluster. The concept of boxes with a resolution is introduced to quantify the precision of the abstraction provided by boxes. The ratio between the number of cells covered by the set of boxes to the total number of cells is used to measure the relative coarseness of the box abstractions. This innovative approach to monitoring neural networks offers a customizable and efficient way to assess monitor effectiveness and precision.

6.2.3 *Active Monitoring*

The paper [40] introduces an innovative algorithmic framework for the active monitoring of neural networks. The primary goal of this monitoring is to maintain accuracy in dynamic environments where inputs frequently deviate from the initially known set of classes. This challenge is addressed by detecting inputs from novel classes and retraining the classifier on an augmented dataset. The framework operates in parallel with the neural network. It interacts with a human user through a series of interpretable labeling queries

for incremental adaptation. Additionally, an adaptive quantitative monitor is proposed to improve precision, confirming the benefits of active monitoring in dynamic scenarios.

The mechanism of monitoring in this framework aims to achieve high precision in detecting novel classes. This goal leaves the run-time performance of the trained model unaffected. It operates in stages, switching between monitoring and adaptation. During monitoring, the monitor reports inputs to the network and submits them to an authority for correct label assignment. Depending on the assessment, the neural network or the monitor is incrementally adjusted, or they are retrained to learn an unknown class. The paper also introduces a quantitative monitor that observes a feature layer and compares its valuation to a model of *typical* behavior for the class. This is achieved through clustering algorithms like k-means, setting reference points for computing a distance function at the cluster centers.

6.3 CONCLUSION

This chapter explores two essential concepts: *test prioritization algorithms* and *runtime monitoring techniques*. The section on test prioritization delves into various techniques like entropy, DeepGini, and others, aiming to uncover potential weaknesses in DNNs and enhance testing efficiency. The runtime monitoring section presents frameworks such as *Outside the Box* and *Active Monitoring*, focusing on the detection of novel or unseen inputs and maintaining accuracy in dynamic environments. These concepts together form a comprehensive approach for improving and understanding neural network testing. This approach emphasizes the shared goal of enhancing the robustness and reliability of DNNs.

In addition, DeepAbstraction++, our recent contribution, overcomes the limitations of the previous work. For instance, it uses runtime monitors and a scoring function to prioritize tests without depending on the DNN's output as DeepGini. This approach is robust, scalable, and precise. Furthermore, it effectively highlights error-prone test instances without relying completely on near-boundary uncertainty but it also includes near-centroid test instances. It recognizes that DNNs can wrongly classify instances with high confidence, independent of their location relative to the decision boundary. This addresses the main limitations of earlier work and establishes a new standard for robust test prioritization in deep learning systems. Eventually, DeepAbstraction++ introduces a much more stable and efficient performance than other state-of-the-art algorithms such as TestRank that requires pre-labeling for some test instances to operate.

Part IV

CONCLUSION & FUTURE WORK

7

CONCLUSION & FUTURE WORK

7.1 CONCLUSION

In conclusion, this thesis addresses important challenges in the test prioritization area. One of the main contributions is a 2-tier prioritization framework, namely, DeepAbstraction. This framework aims to improve test prioritization in deep learning systems by estimating the error-revealing capability of unlabeled test instances. DeepAbstraction is unique in its integration of two pivotal ranking mechanisms: monitors and a scoring function. The monitors play a significant role in detecting error-exposing instances, particularly near the centroids and decision boundaries. Empirical evidence supports the effectiveness of DeepAbstraction, showing that it outperforms other deep learning test prioritization algorithms, including the state-of-the-art TestRank algorithm.

Another novel contribution is the critical examination of existing metrics such as APFD, RAUC, and ATRC, highlighting their limitations in evaluating test prioritization algorithms. Specifically, the research points out that APFD tends to inflate performance evaluations by overlooking labeling costs, while RAUC and ATRC are overly dependent on FDRE rather than FDRO. To address these issues, the research introduces the WFDR, a new metric that assigns weights based on the difficulty of prioritization. Additionally, the study identifies a need for severity-based prioritization, leading to the development of another metric, the SFDR. Empirical tests conducted in the study validate the effectiveness of both WFDR and SFDR, while also revealing the limitations of previously used metrics.

This thesis concludes by addressing a limitation of the DeepAbstraction framework, specifically the challenge of selecting the appropriate τ parameter. A new methodology, known as *combined parameterized boxes*, is introduced to tackle this issue, adding an element of stability to the framework's performance. This methodology uses collective verdicts from monitors with various τ values, assigning weights to ensure balanced decision-making. It also proposes multiple strategies for integrating these weighted verdicts. The enhanced version, DeepAbstraction++, not only outperforms leading algorithms but also improves performance and adds stability to the framework.

7.2 LESSONS LEARNED

The journey of this research has been both challenging and enlightening. It has yielded several key lessons that could guide future endeavors in the realm of Data-Centric AI and test prioritization. Below are these invaluable insights:

- **Data-Centric Approach**: The shift from Model-Centric to Data-Centric AI highlighted the critical role of data quality. Future researchers should not underestimate the importance of data engineering, especially in high-stakes domains like healthcare and autonomous driving.
- **Resource Allocation**: Test prioritization not only helps in early fault detection but also optimizes resource allocation. Future researchers should consider both aspects to develop more efficient and effective test prioritization algorithms.
- **Hyperparameter Sensitivity**: The challenge of selecting the appropriate τ parameter in DeepAbstraction revealed the sensitivity of machine learning models to hyperparameter tuning. This experience underscores the need for robust methodologies to handle such sensitivities.
- **Comprehensive Metric Study**: The intensive study on existing metrics filled a research gap and provided a foundation for the development of WFDR and SFDR. This approach could serve as a model for future research aiming to introduce new metrics.
- **Metric Limitations**: The limitations of existing metrics like APFD, RAUC, and ATRC led to the development of WFDR and SFDR. This experience taught us the importance of questioning established norms and developing better evaluation methods.
- **Balanced Evaluation with WFDR**: The introduction of WFDR offers a more holistic evaluation by considering both fault detection ratio and rate. This balanced approach could guide the development of future metrics.
- **Severity-Based Prioritization**: The introduction of SFDR revealed the importance of considering the severity of faults in test prioritization. This lesson could be particularly valuable for future research in high-stakes domains.
- **Iterative Improvement**: The transition from DeepAbstraction to DeepAbstraction++ taught us the value of iterative research. Future work should embrace this iterative nature, continually seeking to refine and improve upon existing frameworks.
- **Multi-Monitor Decision Making**: The use of multiple monitors with varying tau values in DeepAbstraction++ enhanced the framework's decision-making accuracy. This multi-monitor approach could be a key strategy for future test prioritization frameworks.
- **Unique Weighting System**: The introduction of a unique weighting system to resolve conflicts among different monitor verdicts adds a layer of robustness to the framework. This system could be adapted for other decision-making scenarios in AI.

7.3 FUTURE WORK

In the upcoming sections, we extend the conversation on test prioritization. First, we investigate adversarial attacks and their impact on test prioritization for deep learning models. Next, we delve into how Explainable AI (XAI) can improve both the reliability and understandability of models, specifically through test prioritization. Lastly, we outline the need for developing new test prioritization techniques for specialized and advanced deep-learning tasks. Each subsection aims to shed light on a unique aspect of future research in test prioritization.

7.3.1 *Adversarial Attacks*

An adversarial test instance x' is a perturbed version of a legitimate input x designed to mislead a deep learning model f. The perturbation δ is usually small and often imperceptible to humans, but sufficient to cause the model to make an incorrect prediction or classification. Mathematically, this can be represented as:

$$x' = x + \delta$$

The objective of generating an adversarial instance is to maximize the loss function $J(f(x'), y)$ while keeping δ small. This can be formally defined as an optimization problem:

$$\text{argmax}_\delta \; J(f(x+\delta), y) \quad \text{subject to} \quad ||\delta||_p \leq \epsilon$$

Here, J is the loss function, y is the true label, $||\delta||_p$ is the p-norm of δ, and ϵ is a small constant that constrains the magnitude of the perturbation.

The notation $||\delta||_p$ represents the p-norm of the perturbation δ. In simpler terms, it measures the magnitude or size of the perturbation vector δ using a specific type of norm, denoted by p.

Different values of p yield different types of norms:

- $p = 1$: Manhattan norm (sum of absolute values)
- $p = 2$: Euclidean norm (square root of the sum of squares)
- $p = \infty$: Infinity norm (maximum absolute value)

$$||\delta||_p = \left(\sum_{i=1}^{n} |\delta_i|^p \right)^{1/p} \tag{7.1}$$

Here, δ_i are the individual components of the perturbation vector δ, and n is the dimension of the vector.

Figure 7.1 demonstrates a compelling experiment on the vulnerability of GoogLeNet, a deep learning model, when applied to the ImageNet dataset. Initially, GoogLeNet classifies an image as a *pandas* with a confidence level of 57.7%. The experiment then introduces a subtle but targeted perturbation to the original image. Upon adding this almost imperceptible noise, GoogLeNet dramatically shifts its classification, now identifying the image as a *gibbon* with a strikingly high confidence of 99.3%. Intriguingly, the magnitude of the perturbation, denoted as ϵ, is just 0.007. The experiment vividly illustrates how a

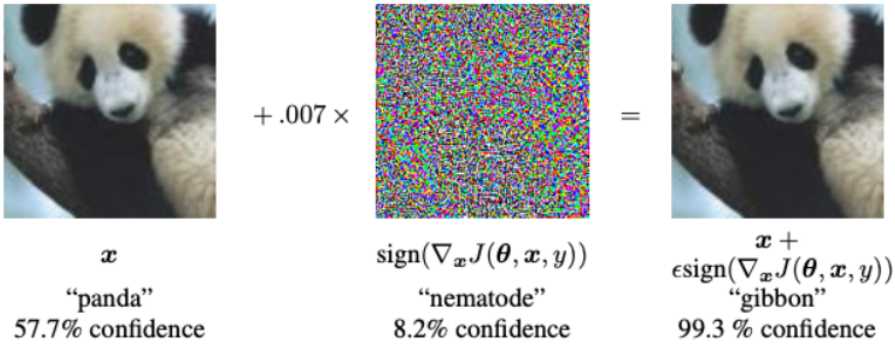


Figure 7.1: Impact of Gradient-Based Perturbations on GoogLeNet with $\epsilon = 0.007$ [25].

minuscule, calculated change can drastically mislead a sophisticated deep learning model into making an incorrect classification.

There are several algorithms to generate these adversarial test instances such as, Fast Gradient Sign Method (FGSM) [25], Carlini & Wagner (C&W) attack [7], Jacobian-based Saliency Map Attack (JSMA) [50], and Projected Gradient Descent (PGD) [44].

7.3.1.1 *Test Prioritization*

The benefits of orienting test prioritization towards adversarial attacks are significant. Firstly, it leads to enhanced model robustness. By identifying and addressing model vulnerabilities early in the development process, we ensure a stronger model foundation. Secondly, this approach can enhance model generalization. Models trained with a focus on adversarial instances are less likely to overfit. They are also more adaptable to diverse input distributions. Moreover, this approach is cost-efficient for model testing. It also strengthens stakeholder confidence. Models that prove their resilience against adversarial attacks are more likely to be trusted and accepted quickly.

After our initial exploration of adversarial attacks, we recognize the importance of evolving our research in test prioritization. In the past, we developed algorithms like DeepAbstraction++ that target misclassified or error-prone instances. However, our vision for the future is broader. We aim to design algorithms that prioritize testing specifically for adversarial instances. Adversarial test generation is crucial in this journey. Adversarial test generation can produce a vast dataset of potential adversarial examples. The sheer size of this dataset, however, is challenging. It is not feasible to test models against every generated instance. Time and computational constraints prevent this. Thus, there is a clear need for adversarial test prioritization algorithms. These algorithms will select the most challenging adversarial examples. This ensures both efficiency in testing and depth of model evaluation.

By integrating adversarial test generation with test prioritization, we are elevating the standards of model evaluation. This approach does not merely assess superficial performance metrics. Instead, it delves deeper, challenging the model's foundational understanding using carefully curated adversarial inputs. Our overarching objective is to cultivate a system that is not only robust and reliable but also adept at navigating the unpredictabilities of real-world data and adeptly handling adversarial challenges.

7.3.2 *Explainable AI*

The following sections delve into XAI, a field that focuses on making the complex mechanisms of deep learning models transparent and understandable. From its importance in test prioritization to various methodologies, we examine how XAI can contribute to making deep learning models both reliable and understandable from the lens of test prioritization.

7.3.2.1 *Overview*

Explainable AI (XAI) is a burgeoning field that aims to make the complex decision-making processes of AI and machine learning models transparent and understandable to human users. The primary objective is to demystify the *black-box* nature of these models, making them more accessible and trustworthy. Thus, XAI empowers users to understand why a model makes a specific decision, thereby increasing confidence in AI systems.

The significance of XAI extends beyond mere curiosity; it has practical implications in various sectors. For instance, in healthcare, understanding why a model recommends a particular treatment can be crucial for patient care. Similarly, in finance, knowing the rationale behind investment decisions can help in risk management. Thus, XAI serves as a bridge between complex AI algorithms and human interpretability.

7.3.2.2 *Contribution of XAI in Test Prioritization*

Test prioritization techniques, such as the DeepAbstraction++ framework, excel at identifying test instances that are likely to reveal errors in AI models. These techniques are invaluable for improving the reliability of AI systems. However, they often lack the capability to explain why certain instances are more error-prone than others, which is where XAI comes into play.

Incorporating XAI into test prioritization can significantly enhance the framework's utility by providing these much-needed explanations. This is particularly important in high-stakes applications like healthcare, finance, and autonomous driving, where understanding the reasoning behind each decision can have serious implications. By offering a *why* along with the *what*, XAI makes these systems not just reliable but also trustworthy.

7.3.2.3 *Global vs. Local Interpretability*

Global interpretability aims to provide a comprehensive understanding of the model's behavior across a wide range of test instances. This is particularly useful for identifying general patterns or rules that the model follows, thereby helping to understand why certain types of instances are more prone to errors than others. However, global interpretability alone may not suffice when delving into the specifics of individual test cases.

Local interpretability, on the other hand, focuses on individual predictions and can offer deep insights into why a particular test instance was flagged as high-risk or error-prone. This level of granularity is crucial for debugging and refining the model. Therefore, a balanced approach that leverages both global and local interpretability could offer the most comprehensive insights into test prioritization.

7.3.2.4 *State-of-the-Art Tools in XAI*

SOTA tools in Explainable Artificial Intelligence (XAI) offer an array of methods to dissect and understand the complex decision-making processes of DNNs. Among the leading tools is Local Interpretable Model-Agnostic Explanations (LIME) [64], which specializes in providing local explanations for individual predictions, allowing users to comprehend how a model arrives at specific conclusions. SHapley Additive exPlanations (SHAP) [41] extends its capabilities to deliver global explanations, which help in understanding the overall behavior of a DNN by shedding light on the relative importance of various input features. Additional methods, e.g., Gradient-weighted Class Activation Mapping (Grad-CAM) [68], produce heatmaps to indicate which parts of an input image, for example, most significantly influence the model's prediction. For instance, Grad-CAM is highly class-discriminative which means it develops a heatmap on the predicted class of the cat and dog for each prediction as shown in Fig 7.2. Integrated Gradients [73] takes this a step further by attributing the contribution of each feature to a particular prediction, thereby offering a fine-grained analysis.

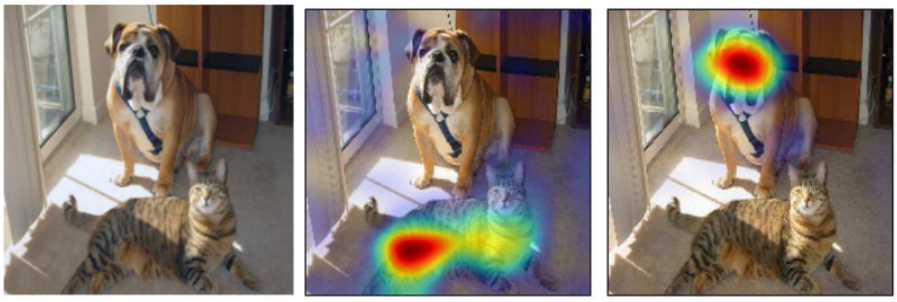


Figure 7.2: GRAD-CAM heatmaps of cat and dog classes from VGG-16 [68].

With this rapid development, researchers are continuously exploring new avenues. Their goal is to increase the effectiveness and efficiency of explanation methods. For the specific purpose of our research, which focuses on test prioritization in DNNs, a strategic combination of these state-of-the-art tools is in planning. This blend aims to capture the best of both local and global interpretability methods. It enables a comprehensive and nuanced understanding of how DNNs make decisions in test prioritization.

7.3.2.5 *Proposed Methodology*

The first step in the proposed methodology involves integrating both local and global interpretability tools into existing test prioritization frameworks like DeepAbstraction++. This will enable a dual capability: identifying high-risk or error-prone test instances and explaining why they are flagged, both on a general and individual level.

The second step involves applying XAI techniques to analyze these prioritized test instances. For global insights, we will focus on understanding the general patterns that lead to errors across multiple instances. For local insights, we will delve into the specifics of individual high-risk test cases. Detailed explanations will be generated for both levels of interpretability. These explanations will then be validated with domain experts to ensure their accuracy and reliability.

7.4 TEST PRIORITIZATION IN ADVANCED DL TASKS

Deep learning has revolutionized numerous industries, introducing complex models that tackle intricate tasks. As we delve deeper, we explore the significance and nuances of advanced tasks like object detection, segmentation, and human pose estimation in the realm of test prioritization.

7.4.1 *Significance of Advanced DL Tasks*

The domains of object detection, segmentation, and human pose estimation remain relatively uncharted in terms of comprehensive test prioritization research. Historically, the bulk of attention has gravitated toward classification-centric endeavors. This asymmetry is particularly striking since real-world applications are now more dependent on these complex deep-learning tasks.

These tasks inherently possess nuanced complexities that render test prioritization challenging. For instance, in object detection, the labeling process is not solely categorizing an image. It mandates annotators to meticulously identify and label every single object in the image. The challenge amplifies when faced with scenes replete with overlapping objects or those spanning varied scales. Moreover, segmentation requires annotators to precisely demarcate boundaries around entities or regions, a task that becomes intricate with overlapping segments or complex textures. Lastly, human pose estimation necessitates pinpointing body parts and their orientations, especially intricate in scenarios with multiple humans or unconventional postures. The painstaking nature of these endeavors escalates labeling costs and time overheads.

7.4.2 *Strategies for Test Prioritization*

For these advanced tasks, test prioritization strategies should emphasize:

- **Object Detection**: Focus on instances with high variability. This includes scenes with multiple object classes, occlusions, or differing lighting conditions. Such a strategy ensures models are resilient against real-world complexities.
- **Segmentation**: Prioritize instances with overlapping segments or intricate textures. This is crucial in domains like medical imaging where discerning subtle differences is paramount.
- **Human Pose Estimation**: Aim for instances involving multiple humans or unconventional postures, ensuring accuracy even in densely populated or uncommon scenarios.

In summary, the emphasis on tasks like object detection, segmentation, and human pose estimation underscores the evolving landscape of deep learning. These tasks present unique challenges. By tailoring test prioritization strategies, we ensure that our models excel in intricate real-world scenarios.

Part V

APPENDIX

8

TECHNICAL DOCUMENTATION

In this chapter, we discuss the technical documentation of our framework. We publicly published the framework's source code in the provided repository.

8.1 FRAMEWORK OVERVIEW

DeepAbstraction is a test prioritization framework designed to minimize labeling costs in Deep Neural Networks (DNNs) while efficiently identifying corner cases. The framework leverages abstraction-based monitors to assign verdicts to test instances. DeepAbstraction excels in identifying corner cases in both near-boundary and near-centroid regions of the feature space. This framework shows better performance than the current state-of-the-art techniques.

8.2 FRAMEWORK ARCHITECTURE

This section details the framework's main components and operational workflow.

8.2.1 *Components*

The DeepAbstraction framework is composed of several interlinked components, each serving a specific role in test prioritization.

1. **Deep Neural Network (DNN)**: Serves as the backbone for predictions. The DNN processes both training and test datasets.
2. **Runtime Monitors**: Supervise the decision-making of the DNN. They provide three types of verdicts for each test instance: acceptance, uncertainty, or rejection. These monitors are built using high-level features from the DNN and are key in creating box abstractions.
3. **Box Abstractions**: Generated post-training, these multi-dimensional boxes encapsulate instances sharing similar high-level features. They are constructed using two types of subsets—correctly classified and incorrectly classified—based on the DNN's predictions.
4. **Statistical Scoring Functions**: Utilized for quantifying the error-revealing capability of each test instance. The framework employs Gini Impurity and Entropy as the scoring functions.

8.2.2 *Workflow*

The framework's operation is divided into several phases, each contributing to effective test prioritization.

1. **Training Phase**: The DNN is trained using the training dataset, and high-level features are collected from the penultimate layer for each training instance.
2. **Monitor Construction**: Post-training, runtime monitors are built using the collected high-level features and their corresponding predicted classes. Box abstractions are then constructed.
3. **Test Phase**: The DNN processes the test dataset, and runtime monitors provide verdicts based on which box abstractions the test instances fall into.
4. **Evaluation and Scoring**: Finally, test instances are scored using statistical scoring functions, prioritizing them based on their likelihood to reveal errors.

8.3 SYSTEM REQUIREMENTS

To successfully run the DeepAbstraction framework, certain hardware and software specifications are required. The following sections outline these prerequisites.

8.3.1 *Hardware Requirements*

A robust hardware setup ensures the smooth operation of DeepAbstraction. The minimum requirements include:

- CPU: Intel Core i5 or equivalent.
- RAM: 8 GB minimum.
- Disk Space: 20 GB free space.
- GPU: NVIDIA GeForce GTX 1050 or equivalent (optional but recommended for accelerated computation).

8.3.2 *Software Requirements*

Compatibility and performance also depend on the software environment. The framework has been tested under the following conditions:

- Operating System: Ubuntu 18.04 or higher, Windows 10, or macOS Catalina or higher.
- Python: Version 3.7 or higher.
- Libraries: NumPy, PyTorch, and other dependencies as listed in the `requirements.txt` file.

8.3.3 *Additional Tools*

Additional software tools enhance the framework's functionality and user experience. Currently, the framework requires:

- Jupyter Notebook or Jupyter Lab: For running the provided notebook files.

8.4 REPOSITORY ARCHITECTURE

The DeepAbstraction framework's repository is organized to ensure easy navigation and usage. This section outlines the structure and purpose of each file and directory.

8.4.1 *Directory Structure*

The repository contains several directories, each housing specific types of files for the framework.

1. **Notebooks**: This directory holds all Jupyter Notebook files, which include `training_and_test.ipynb`, `processing.ipynb`, and `analysis_and_evaluation.ipynb`. We will explain each notebook later.
2. **Data**: stored within `npz_data` folder, each NPZ-formatted file represents a dataset. These files contain high-level features, labels, and predicted classes for the training dataset. Additionally, they include high-level features, labels, predicted classes, and the Gini index for each test instance. The `monitors_data` folder contains an Excel file for each dataset, storing scoring functions such as Gini and entropy, and also houses the verdicts from each monitor.

3. **Analysis**: This directory `tau_effect` contains analytical outcomes for each dataset. Individual files within this directory illustrate the impact of the clustering parameter (τ) on various groups formed by the monitors' verdicts. These groups include True Positives, False Positives, Uncertain Positives, Uncertain Negatives, True Negatives, and False Negatives.

8.4.2 *Key Files*

The framework relies on several pivotal files that facilitate its operation.

1. **Toolkit**: This directory contains essential scripts for building abstraction-based monitors, including those that handle the clustering step. It also has scripts that execute the monitors, render verdicts for each test instance, and store these verdicts into an Excel file.
2. **requirements.txt**: list the Python libraries necessary for the successful execution of the framework.
3. **readme.md**: Furnishes a comprehensive overview and setup guidelines for users to navigate and operate the DeepAbstraction framework effectively.
4. **overall performance.xlsx**: An Excel spreadsheet that stores analytical evaluations of DeepAbstraction's performance across different values of the clustering parameter(τ).

8.5 CONTRIBUTION 1: DEEPABSTRACTION FRAMEWORK

8.5.1 *Training and Testing Notebook*

The `training_and_test.ipynb` notebook focuses on training and testing deep learning models using PyTorch. It covers various aspects such as importing libraries, setting hyperparameters, selecting models and datasets, and running the training and test phases.

8.5.1.1 *Import Libraries and Setup*

This section imports essential libraries for data manipulation, visualization, and deep learning. The section also sets up the notebook environment for inline plotting and suppresses warning messages for cleaner output.

```
import os
import warnings
import numpy as np
...
```

8.5.1.2 *Hyperparameters*

Hyperparameters for the training process are set in this section. These hyperparameters are crucial for the training process and can be tuned for better performance. The user can change N_CLASSES according to the number of classes in the selected dataset.

```
n_epochs = 3
batch_size = 64
n_classes = 10
```

8.5.1.3 *Model Selection*

The `ModelManager` class offers a seamless approach for selecting and configuring pre-trained models. Specifically, the class adjusts the output layer to align with the number of classes in the dataset, enhancing its adaptability for diverse tasks. For a broader range of pre-trained model options, users can consult this PyTorch Models page.

```
class ModelManager(nn.Module):
    def __init__(self):
        super(ModelManager, self).__init__()
            ...
```

8.5.1.4 *Dataset Selection*

The `CustomDataset` class provides a flexible framework for dataset selection and configuration. Users can easily adapt this class to suit various dataset types. For additional dataset options, users can refer to this PyTorch Datasets page. It is important to note that changing the dataset may necessitate adjustments to the architecture of the chosen model, especially for the last layer, which must align with the number of classes in the dataset.

```
class CustomDataset(Dataset):
    def __init__(self, dataset_dir, dataset_name, train=True, transform=None):
        ...
    def __len__(self):
        ...
    def __getitem__(self, index):
        ...
    def display(self, index):
        ...
```

The CustomDataset class serves as a customizable interface for loading and manipulating datasets. Below are the functions defined within this class:

- `__init__`: Initializes the dataset and sets up any required transformations.
- `__len__`: Returns the total number of samples in the dataset.
- `__getitem__`: Retrieves a sample from the dataset at a specified index and applies any set transformations.
- `display`: Displays an image from the dataset at a specified index along with its label.

8.5.1.5 *Training and Test Phases*

The `Manager` class serves as a comprehensive utility for managing the training and testing phases of machine learning models. It handles data loading, model optimization, and

performance evaluation, among other tasks. Below are the functions defined within this class:

```
class Manager:
    def __init__(self, model, train_loader, test_loader, optimizer, criterion,
    device):
        ...
    def train(self, epoch):
        ...
    def test(self):
        ...
    def run(self, n_epochs):
        ...
```

- `__init__`: Initializes the manager with the model, data loaders, optimizer, loss criterion, and device information.
- `train`: Executes the training phase for one epoch, updating the model parameters.
- `test`: Evaluates the model on the test dataset and calculates performance metrics.
- `run`: Orchestrates the training and testing phases over multiple epochs.

8.5.1.6 *Run Training and Testing*

This section initializes the model, dataset, and other components, and then runs the training and testing phases. This is where the actual computation happens.

```
manager = Manager()
manager.run(n_epochs)
manager.evaluate()
```

8.5.1.7 *Storing the Data*

Upon completing the training and testing phases, the notebook saves various types of results into a compressed NumPy file to be the input of the next notebook `processing.ipynb`. This file encapsulates a wide range of data, including:

```
np.savez(...)
```

- `train_logits` and `test_logits`: The logits for the training and test datasets.
- `mis_test_indices`: Indices of the test samples that were misclassified.
- `train_pred_labels` and `test_pred_labels`: The predicted labels for the training and test datasets.
- `train_gt_labels` and `test_gt_labels`: The ground-truth labels for the training and test datasets.
- `gini_score`: The Gini scores for test instances.

8.5.1.8 *Visualize Some Samples*

The notebook displays some sample images from the dataset. This provides a visual understanding of the kind of data the model has been trained on.

```
test_dataset.display(10)
train_dataset.display(6666)
```

8.5.2 *Processing Notebook*

This notebook focuses on processing and analyzing the data generated from the training and testing phases. It covers the following main sections:

8.5.2.1 *Import Libraries & Create Folders*

Importing necessary libraries and setting up directories for storing various types of data.

```
import os, csv, glob, time, warnings, ...
warnings.filterwarnings('ignore')
```

8.5.2.2 *Dataset Selection*

The `DatasetLoader` class serves as a utility for dataset selection and management. It lists all available datasets in a specified directory and allows the user to choose one for the experiment. The class is designed to be highly flexible and adaptable, making it easy to work with various types of datasets.

```
class DatasetLoader:
    def __init__(self, dataset_dir):
        ...
    def _list_datasets(self):
        ...
    def _create_dataframe(self):
        ...
    def select_dataset(self):
        ...
```

- `__init__`: Initializes the class and lists available datasets by invoking the `_list_datasets` method.
- `_list_datasets`: A private method that lists all datasets available in the given directory.
- `_create_dataframe`: Creates a DataFrame to display the available datasets, aiding the user in making a selection.
- `select_dataset`: Prompts the user to select a dataset and returns the selected dataset.

8.5.2.3 *Data Export to CSV*

This section focuses on exporting the high-level features of the data into CSV files for both the training and testing phases. The data is categorized based on whether the prediction was correct or not, and then saved into separate CSV files for each class.

```
# Define header and data lists
header = ['index', 'Ground Truth', 'Predicted class', ...]
    ...

# Function to write CSV
def write_csv(name, rows, header):
    ...

# Loop through data types (training/testing)
for folder, data_list in data_types.items():
    ...
    for i in range(num_classes):
        ...
        write_csv(f'{path}/class_{i}_good_high_level_features.csv', final_good,
    header)
        write_csv(f'{path}/class_{i}_bad_high_level_features.csv', final_bad,
    header)
```

8.5.2.4 *Data Preparation and Monitor Construction*

This section covers both the preparation of data for monitor construction and the actual construction of the monitors. The data is formatted to be suitable for monitor construction, and then the `construct_monitors` function is used to build the monitors.

```
# Prepare Data for Monitor Construction
num_classes = 10
data = np.load(...)

# Monitor Construction
construct_monitors(experiment_name, tau_list, range(num_classes))
```

8.5.2.5 *Stable Softmax Functions*

This function computes the softmax in a numerically stable manner. It is often used as a preliminary step before calculating other uncertainty measures like Gini Impurity and Entropy.

```
# Function to compute stable softmax
def stable_softmax(x):
    ...
```

8.5.2.6 *Uncertainty measures and Distance Ratio Computation*

This section is dedicated to computing the uncertainty measures and distance ratios for test samples. Uncertainty measures such as Deep Gini and entropy are calculated for each test sample. Additionally, the distance ratio is computed based on the Euclidean distance to the center of each class, derived from the training data.

```
# Initialize variables and header
final_features = []
header = ['index', 'Ground Truth', 'Predicted class', 'deepgini', 'entropy']
...

# Collect features for each test sample
for j in range(num_test_samples):
    ...
    final_features.append(row_data)

# Initialize DataFrame and distance ratio
df = pd.DataFrame(final_features, columns=header)
df["distance_ratio"] = -1
...

# Compute accumulated sum and center for each class
acc_sum = np.zeros((num_neurons, num_classes))
center_per_class = np.zeros((num_neurons, num_classes))
...

# Compute distance ratios for test samples
for i in range(test.shape[1]):
    ...
    df.loc[i, "distance_ratio"] = dist_to_pred / dist_to_gt

df.head()
```

8.5.2.7 *DataFrame Aggregation and Excel Export*

This section is responsible for aggregating multiple DataFrames and exporting the final DataFrame to an Excel file. The DataFrames are read from a directory and concatenated. The final DataFrame is then merged with an existing DataFrame, sorted, and exported to an Excel file with specific formatting.

The following key operations are performed in this code block:

- `file_dfs` and `fields_to_read`: Initialize the list for storing individual DataFrames and specify the fields to read from each CSV file.
- `Read and append individual DataFrames`: Loop through the directory to read each CSV file and append it to the list.
- `Concatenate and merge DataFrames`: Concatenate the list of DataFrames and merge it with an existing DataFrame.

- `Write the final DataFrame to Excel`: Export the final, merged DataFrame to an Excel file, applying specific formatting like freezing panes and setting column widths.

```
# Initialize list for DataFrames and fields to read
file_dfs = []
fields_to_read = [...]
...

# Read and append individual DataFrames
for csv_file in os.listdir(verdict_dir):
    ...
    file_dfs.append(temp_df)

# Concatenate and merge DataFrames
concatenated_df = pd.concat(file_dfs).sort_values(by=["index"])
merged_df = pd.merge(df, concatenated_df, ...)
merged_df.set_index('index', inplace=True)

# Write the final DataFrame to Excel
with pd.ExcelWriter(excel_path, engine='xlsxwriter') as writer:
    ...
    merged_df.to_excel(writer, ...)
    ...
```

8.5.2.8 *Verdict Monitors Count*

This section focuses on calculating different types of verdict counts for various τ values. The verdict counts include True Positives (TP), False Positives (FP), Uncertain Positives (UP), Uncertain Negatives (UN), False Negatives (FN), and True Negatives (TN). These counts are stored in a DataFrame indexed by τ values.

```
# Initialize DataFrame for monitor verdicts
tau_metrics_df = pd.DataFrame(index=tau_list, columns=['TP', 'FP', 'UP', 'UN', 'FN
    ', 'TN'])
...

# Loop through tau values to count verdict types
for tau in tau_list:
    verdict_col = f'verdict_{tau}'
    ...
    tau_metrics_df.loc[tau, metric] = count_verdict_types(verdict_col, condition)
```

The code performs the following key operations:

- `tau_metrics_df`: Initializes a DataFrame to store the counts of different verdict types for each τ value.
- `count_verdict_types`: A helper function that counts the number of instances that meet a specific condition for a given verdict column.

- `Loop through τ values`: Iterates through the list of τ values, calculating the counts for each verdict type and storing them in the DataFrame.

8.5.3 *Analysis and Evaluation Notebook*

This notebook serves as the final step in the pipeline, focusing on the analysis and evaluation of the test prioritization techniques. It not only assesses the effectiveness of various test prioritization strategies but also offers a visual interpretation through graphs and charts.

8.5.3.1 *Import Libraries and Initialize Variables*

This section is responsible for importing all the necessary Python libraries and initializing variables that will be used throughout the notebook. These libraries are pivotal for data analysis, visualization, and the computation of various performance metrics. Additionally, variables that will be used throughout the notebook for storing paths, filenames, and other configuration settings are initialized here.

```
import pandas as pd
import numpy as np
...

# Initialize variables
data_path = "path/to/data"
output_dir = "path/to/output"
...
```

8.5.3.2 *Effectiveness Evaluation*

This section aims to evaluate the effectiveness of various scoring functions. It includes the computation of metrics like True Positives, False Positives, and so on.

```
def compute_metrics(df):
    ...
```

Furthermore, this section is dedicated to evaluating the effectiveness of various test prioritization techniques: DeepGini, Entropy, and DeepAbstraction. It calculates the TPF for each technique and uses these TPF values to compute the Average Test Percentage of Fault (ATPF) to evaluate the effectiveness of each prioritization algorithm.

```
# Function to calculate TPF
def calculate_TPF(df, no_buggy_instances):
    ...
# Loop through each dataset
for dataset in datasets:
    ...
    # DeepGini
    df.sort_values(by=['deepgini'], ascending=False, inplace=True)
    gini = calculate_TPF(df, no_buggy_instances)
    ...
    # Entropy
    df.sort_values(by=['entropy'], ascending=False, inplace=True)
    entropy = calculate_TPF(df, no_buggy_instances)
    ...
    # DeepAbstraction
    for tau in tau_list:
        ...
        df_monitor = pd.concat(dfs_sorted)
        ...
        tpf_monitor = calculate_TPF(df_monitor, no_buggy_instances)
        ...
        # Update arr_ATFP
        arr_ATFP[row, 4 + methods.index(method)] = round(100 * sum(tpf_monitor) /
    no_buggy_instances, 2)
    ...
```

8.5.3.3 *Zero-Scored Instances Removal Effect*

This section performs a multi-step analysis that starts by initializing arrays for storing dataset-specific information and defining a helper function for updating zero-scored instances. It then preprocesses the data by sorting the DataFrame based on DeepGini scores and creating new DataFrames based on verdicts. Following this, a DataFrame is created to summarize the zero-scored instances. Finally, the section concludes with a data visualization step, where a bar chart is generated using Plotly to visualize the distribution of True Positive and False Positive instances before and after the removal of zero-scored instances.

The code performs the following key operations:

- `Initialization and Helper Function`: Arrays are initialized for storing dataset-specific information, and a helper function is defined for updating zero-scored instances.
- `Data Preprocessing`: The DataFrame is sorted, and new DataFrames are created based on verdicts, which are then used to update the initialized arrays.
- `Summary and Visualization`: A DataFrame summarizing zero-scored instances is created, followed by a bar chart that visualizes the distribution of instances.

```
# Initialize arrays
arr_miss = np.zeros((len(datasets), 7), dtype=object)
arr_zero_remov = np.zeros((len(datasets) * 2, 4), dtype=object)

# Function to update array for zero-scored instances
def update_zero_remov_array(arr, idx, dataset, state, df, verdict, j):
...

# Create DataFrame from arr_zero_remov and display the first 20 rows
zero_removal_df = pd.DataFrame(
    arr_zero_remov, columns=['dataset', 'when', 'TP', 'FP']
)
...

# Melt the DataFrame to a long format suitable for plotting
condition = zero_removal_df['dataset'].str.contains('ResNet18')
...

# Create a bar chart using Plotly
fig = px.bar( new_df, x='when', y='value', facet_col='dataset',
...
```

8.5.3.4 *Stability Evaluation*

This section is dedicated to evaluating the stability of the DeepAbstraction technique using different scoring functions: Gini Index and Entropy. Two line charts are generated using Plotly, each representing the Average Test Percentage of Fault (ATPF) against different values of the clustering parameter (τ). Both figures are saved as image files for future reference and are displayed in the notebook using the Matplotlib library.

```
# line chart (Gini Index)
fig = px.line(
    final_df, x='tau', y='DeepAbstr_gini', color='dataset',
    ...
)
pio.write_image(fig, f'{img_dir}DA_gini.png', scale=6)

# line chart (Entropy)
fig = px.line(
    final_df, x='tau', y='DeepAbstr_entropy', color='dataset',
    ...
)
pio.write_image(fig, f'{img_dir}DA_entropy.png', scale=6)
```

8.6 CONTRIBUTION 2: EVALUATION METRICS

In this paper, we do not repeat the work presented in the first two notebooks of the first paper. Instead, we aim to evaluate test prioritization techniques such as DeepGini, Neurons

Pattern, and DeepAbstraction using recently developed metrics introduced in the second paper. Therefore, we proceed directly to evaluation and subsequent results visualization.

8.6.1 *Analysis and Evaluation Notebook*

This notebook serves as a comprehensive tool for evaluating test prioritization techniques across multiple datasets. It dives into the effectiveness evaluation of various techniques, applying several metrics like APFD, RAUC, ATRC, WFDR, and SFDR for assessment. Finally, the notebook performs statistical analysis and saves the results for the visualization notebook.

8.6.1.1 *Import Libraries*

This section imports all the essential Python libraries that will be used throughout the notebook.

```
import os
import numpy as np
import pandas as pd
...
```

8.6.1.2 *Configuration Parameters*

This section sets up configuration parameters and variable initialization that are crucial for the notebook. These include the list of τ values and the directory where the data resides.

```
tau_list = [0.1, 0.05, 0.1, 0.01, 0.4, 0.4, 0.1, 0.05]
data_dir = "./monitors data/"
...
```

8.6.1.3 *Evaluation Metrics*

This section is dedicated to defining various evaluation metrics that will be used for assessing the effectiveness of test prioritization techniques. The metrics include APFD, RAUC, ATRC, WFDR, and SFDR. These metrics collectively provide a comprehensive evaluation of the test prioritization techniques.

```
def APFD(lst, n):
    ...
def RAUC(lst, n):
    ...
def ATRC(lst, n):
    ...
def WFDR(lst, n):
    ...
def SFDR(lst, n):
    ...
```

8.6.1.4 *Effectiveness Evaluation*

This section evaluates the effectiveness of various techniques on multiple datasets. It reads each dataset, applies the techniques, and stores the results in a DataFrame.

```
eval_df = pd.DataFrame()
for i in range(len(datasets)):
    df = pd.read_excel(f'{data_dir}{datasets[i]}.xlsx', sheet_name=0)
    ...
    # DeepGini
    df.sort_values(by=["deepgini"], ascending=False, inplace=True)
    gini = list(df['Ground Truth'] != df['Predicted class'])[:no_bugs]
    ...

    # Neurons Pattern
    df.sort_values(by=["neuron_pattern"], inplace=True)
    pattern = list(df['Ground Truth'] != df['Predicted class'])[:no_bugs]
    ...

    # DeepAbstraction
    j = tau_list[i]
    df_monitor1 = df[df[f'verdict_{j}'] == 1 ] # rejection group
    df_monitor2 = df[df[f'verdict_{j}'] == 2 ] # uncertain group
    df_monitor3 = df[df[f'verdict_{j}'] == 0 ] # acceptance group
    ...
```

8.6.1.5 *Save Evaluation Data*

This section saves the evaluation data to a parquet file for the visualization notebook. The parquet format is efficient for storing large DataFrames.

```
eval_df.to_parquet('eval_df.parquet', engine='fastparquet')
```

8.6.1.6 *Statistical Analysis*

This section conducts a thorough statistical analysis on the evaluation data, storing the results in three distinct dataframes. The first dataframe, `stat_df1`, quantifies the fault detection ratio across multiple test prioritization algorithms. The second and third dataframes, `stat_df2` and `stat_df3`, break down the fault detection ratio for each algorithm according to severity levels.

```
stat_df1 = pd.DataFrame()
stat_df2 = pd.DataFrame()
stat_df3 = pd.DataFrame()

thrshld1 = 50
thrshld2 = 80

for i in range(len(datasets)):

    df = pd.read_excel(f'{data_dir}{datasets[i]}.xlsx', sheet_name=0)
    no_bugs = sum(df['Ground Truth'] != df['Predicted class'])
    ...

    # DeepGini
    df.sort_values(by=['deepgini'], ascending=False, inplace=True)
    gini_sum = sum(df.iloc[:no_bugs,0] != df.iloc[:no_bugs,1])
    ...

    # Neurons Pattern
    df.sort_values(by=["neuron_pattern"], inplace=True)
    pattern_sum = sum(df.iloc[:no_bugs,0] != df.iloc[:no_bugs,1])
    ...

    #DeepAbstraction
    j = tau_list[i]
    df_monitor1 = df[df[f'verdict_{j}'] == 1 ] # rejection group
    df_monitor2 = df[df[f'verdict_{j}'] == 2 ] # uncertain group
    df_monitor3 = df[df[f'verdict_{j}'] == 0 ] # acceptance group
    ...
```

8.6.2 *Data Visualization Notebook*

This notebook serves as a visualization tool for the evaluation results of various test prioritization techniques. It employs Plotly to generate polar plots, bar charts, and histograms, representing metrics like WFDR and SFDR. The visualizations are saved as image files for future reference.

8.6.2.1 *Import Libraries and Initialize Variables*

This section imports all the essential Python libraries and initializes variables that will be used throughout the notebook.

```
import os
import pandas as pd
import plotly.express as px
...
```

8.6.2.2 *Highlight Metrics*

This section defines functions to highlight specific metrics like WFDR and SFDR in the imported DataFrame from the previous notebook. It then applies these functions to color-code the metrics among other metrics.

```
def highlight_metric1(s):
    color = 'blue'
    ...
def highlight_metric2(s):
    color = 'red'
    ...
```

8.6.2.3 *Metrics Evaluation*

This section generates polar plots to evaluate different prioritization metrics like APFD, RAUC, ATRC, WFDR, and SFDR on various experiments. The layout is customized, and the figure is saved as an image.

```
colors = ["#7570B3", "#D95F02", "#30A784","#44bcd8","#E7298A"]
...
for i, approach in enumerate(approaches,1):
    for j, name in enumerate(metrics):
        data[j] = df.loc[df.metric == name, approach].tolist()
        data[j].append(data[j][0])
        fig.add_trace(go.Scatterpolar(r=data[j], theta=datasets,
        ...

fig.show()
pio.write_image(fig, 'Metrics_Evaluation.png',scale=6, width=1200, height=500)
```

8.6.2.4 *Algorithms Evaluation*

This section generates a bar chart to evaluate the performance of different test prioritization techniques such as DeepGini, DeepAbstraction, and Neurons Pattern using WFDR and SFDR metrics. The layout is customized, and the figure is saved as an image.

```
approach_colors = {
    'DeepGini': '#A31414',
    'DeepAbstraction': '#692063',
    'Neurons Pattern': '#00446F'
}
...
bar_chart.show()
pio.write_image(bar_chart, 'Algorithms_Evaluation.png', scale=6)
```

8.6.2.5 *Analysis of Severity Levels Across Datasets*

This section investigates the significance of severity prioritization by evaluating the ratio of high-severity instances in relation to the overall count of misclassified instances. It reads and preprocesses each dataset, categorizes probabilities into labels, and then creates a histogram plot to visualize the distribution of different levels of severity across datasets.

```
# Define color mapping for probability ranges
probability_color_map = {
    '0-20': '#FF6B6B',
    '20-40': '#5F6573',
    '40-60': '#00A885',
    '60-80': '#04607C',
    '80-100': '#840651'
}
...
# Create a histogram plot
histogram_fig = px.histogram(
    renamed_data,
    x='dataset',
    y='probability',
    color='Label',
    ...
)
histogram_fig.show()
```

8.7 CONTRIBUTION 3: DEEPABSTRACTION++ FRAMEWORK

This paper improves the performance of the first paper (DeepAbstraction). More particularly, this paper solves the problem of instability of the performance in DeepAbstraction. The user can replicate the same training and testing phases, and monitor construction from the first paper.

8.7.1 *DeepAbstraction++ Notebook*

This notebook performs a detailed evaluation of test prioritization techniques, specifically focusing on the DeepAbstraction++ algorithm. Therefore, this notebook aims to combine the verdicts of the monitors strategically and prioritize test instances. It employs a variety of metrics for a thorough assessment such as WFDR and SFDR and saves the evaluation data for subsequent visualization.

8.7.1.1 *Import Libraries*

This section imports essential Python libraries required.

```
import pandas as pd
import numpy as np
...
```

8.7.1.2 *Evaluation Metrics*

This section is dedicated to defining various evaluation metrics that will be used for assessing the effectiveness of test prioritization techniques. The metrics include WFDR (Weighted Fault Detection Ratio) and SFDR (Severity Fault Detection Rate). These metrics collectively provide a comprehensive evaluation of the test prioritization techniques.

```
def WFDR(lst, size, n=1):
    """Calculate the Weighted Fault Detection Ratio."""
    ...
def SFDR(ideal, actual):
    """Calculate the Severity Fault Detection Rate."""
    ...
```

8.7.1.3 *Initialization and Settings*

This section initializes important directories and lists the datasets that will be used in the notebook. It also sets up an array for storing metrics values for different methods.

```
DATA_DIR = "./monitors_data/"
IMG_DIR = "./images/"
os.makedirs(IMG_DIR, exist_ok=True)
...
```

8.7.1.4 *DeepAbstraction++ Evaluation*

This section presents the final evaluation of DeepAbstraction++ using two key metrics: Weighted Fault Detection Ratio (WFDR) and Severity Fault Detection Rate (SFDR). The evaluation process iterates through each dataset, applies the DeepAbstraction++ technique, and stores the results in an array named `arr`.

```
row = 0
arr = np.zeros((num_datasets, 3), dtype=object)

for dataset in datasets:
    dataset_path = f"{DATA_DIR}{dataset}.xlsx"
    ...
    # First metric: WFDR
    df["score"] = df[columns_to_update].mean(axis=1)
    ...
    # Second metric: SFDR
    severe_ideal = df.query('(`Ground Truth` != `Predicted class`)')["probability"
    ] \
                    .sort_values(ascending=False).tolist()
    ...
    arr[row, 0] = dataset
    arr[row, 1] = WFDR(monitor, len(df))
    arr[row, 2] = SFDR(severe_ideal, severe_monitor)

    row += 1

    print(f"DeepAbstraction++ has been evaluated successfully on {dataset}")

df = pd.DataFrame(arr, columns=["experiment", "WFDR", "SFDR"])
```

8.7.1.5 *Verdict Type Impact Analysis*

This part of the notebook investigates how different verdict types —acceptance, uncertainty, and rejection — affect the final decision. The code calculates the proportion of misclassified instances for each verdict type when all six monitors agree on the verdict.

```
row = 0
results_df = pd.DataFrame(
    columns=["dataset", "model", "verdict_type", "misclassified", "total"]
)
...
# Loop through datasets and verdict types
for dataset in datasets:
    ...
    for name, verdict in dict_verdict.items():
        ...
        results_df.loc[row] = [
            dataset.split('-')[0], dataset.split('-')[1], name,
            num_verdict, total_num_verdict
        ]
        row += 1
    print(f"All TP techniques have been evaluated successfully on {dataset}")

results_df["proportion"] = results_df["misclassified"] / results_df["total"]
results_df["proportion"] = results_df["proportion"].round(2)
...
```

8.7.1.6 *Plotting the Impact*

The code also includes a plot to visualize the impact of each verdict type on the final decision.

```
fig = go.Figure()
...
# Create a box plot for the distribution of proportions
for idx, verdict in enumerate(results_df["verdict_type"].unique()):
    ...
    fig.add_trace(
        go.Box(
            y=df_verdict["proportion"], name=verdict, boxpoints='all',
            marker_color=colors[idx]
        )
    )
...
fig.show()
```

The analysis reveals that the rejection verdicts have a significant impact on the final decision, outperforming the other verdict types in most experiments.

8.7.1.7 *Impact of Number of Rejection Verdicts*

This section of the notebook examines how the number of rejection verdicts influences the final decision. The code calculates the proportion of misclassified instances for different numbers of rejection verdicts (1, 3, 6) across multiple experiments.

```
row = 0
results_df = pd.DataFrame(
    columns=["No.Exp", "num_reject", "misclassified", "total"]
)
...
# Loop through datasets and number of rejection verdicts
for idx, dataset in enumerate(datasets):
    ...
    for num in [1, 3, 6]:
        ...
        results_df.loc[row] = [
            "Exp." + str(idx), num, num_verdict, total_num_verdict
        ]
        row += 1
    print(f"All TP techniques have been evaluated successfully on {dataset}")

results_df["proportion"] = results_df["misclassified"] / results_df["total"]
results_df["proportion"] = results_df["proportion"].round(2)
...
```

8.7.1.8 *Visualizing the Impact of Rejection Numbers*

The code also includes a bar chart to visualize the impact of different numbers of rejection verdicts on the final decision.

```
colors = {'1': '#5F6573', '3': '#04607C', '6': '#840651'}
...
# Create a bar chart for the distribution of proportions
for reject_val in results_df['num_reject'].unique():
    ...
    trace = go.Bar(
        name=str(reject_val), x=df['No.Exp'], y=df['proportion'],
        marker_color=color
    )
    data.append(trace)
...
fig.show()
```

The analysis shows that the number of rejection verdicts plays a crucial role in the final decision. The proportion of misclassified instances increases as the number of rejection verdicts rises, emphasizing the importance of rejection verdicts in the decision-making process.

8.7.1.9 *Comparative Study on Algorithm Effectiveness*

This section of the notebook evaluates the effectiveness of three algorithms: DeepGini, DeepAbstraction, and DeepAbstraction++. The code uses the WFDR metric to compare these algorithms across eight experiments. Different values of τ are also considered for each experiment.

```
tau_list = [0.05, 0.05, 0.05, 0.05, 0.4, 0.4, 0.05, 0.05]
arr = np.zeros((num_datasets, 4), dtype=object)
row = 0
...
# Loop through datasets and tau values
for idx, (dataset, t) in enumerate(zip(datasets, tau_list)):
    ...
    arr[row, 0] = "Exp." + str(idx)
    arr[row, 1] = WFDR(gini_lst, len(df))
    arr[row, 2] = WFDR(wfdr_lst, len(df))
    arr[row, 3] = WFDR(monitor_lst, len(df))
    row += 1
    print(f"All TP techniques have been evaluated successfully on {dataset}")
```

8.7.1.10 *Tabulating the Results*

The code then tabulates these results into a DataFrame for easier interpretation and comparison.

```
col_names = ["Experiment","DeepGini","DeepAbstraction", "DeepAbstraction++"]
df = pd.DataFrame(arr, columns = col_names)
```

8.7.1.11 *Performance Stability of DeepAbstraction++*

This section evaluates the stability of the DeepAbstraction++ model by varying the β parameter. The code calculates the WFDR metric for each β value, ranging from 1 to 5000, across multiple datasets.

```
row = 0
arr = np.zeros((num_datasets, 3), dtype=object)
...
# Loop through datasets
for idx, dataset in enumerate(datasets):
    ...
    wfdr_lst = []
    for beta in range(1, 5100, 200):
        ...
        wfdr_lst.append(WFDR(monitor_lst, len(df)))
    ...
    arr[row, 0] = "Exp." + str(idx)
    arr[row, 1] = list(range(1, 5100, 200))
    arr[row, 2] = wfdr_lst
    row += 1
    print(f"All TP techniques have been evaluated successfully on {dataset}")
```

8.7.1.12 *Stability Visualization*

To visualize the stability in performance across different β values, we use the Plotly library. The code snippet below shows how to create a line plot for each experiment.

```
fig = go.Figure()
colors = [
    '#d55c93', '#236ccc', '#19d3f3', '#b78b7b',
    '#28d2a3', '#630fff', '#ef573d', '#ab63fa'
]
# Loop through each experiment and add a line trace
for i, exp in enumerate(data_expanded['Experiment'].unique()):
    exp_data = data_expanded[data_expanded['Experiment'] == exp]
    fig.add_trace(
        go.Scatter(
            x=exp_data['Beta'],
            y=exp_data['Performance'],
            ...
        )
    )
...
fig.show()
```

8.7.1.13 *Effectiveness of Different Combination Strategies*

We evaluate the effectiveness of different combination strategies, namely mean, max, product, and mode, using the WFDR metric. This section explains how the code evaluates the effectiveness of different combination strategies. It also discusses the insights gained from the evaluation, highlighting the strengths and weaknesses of each strategy.

```
row = 0
strategies = ['mean', 'max', 'Product', 'mode']
arr = np.zeros((num_datasets * len(strategies), 3), dtype=object)
...
df = pd.DataFrame(arr, columns=["Experiment", "Strategy", "Performance"])
```

The dataFrame is then reshaped for easier analysis.

```
reshaped_df = df.pivot(index='Experiment', columns='Strategy', values='Performance
    ')
reshaped_df = reshaped_df[['mean', 'Product', 'mode', 'max']]
reshaped_df.reset_index(inplace=True)
```

BIBLIOGRAPHY

[1] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. "The k-means algorithm: A comprehensive survey and performance evaluation." In: *Electronics* 9.8 (2020), p. 1295.

[2] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. "End to end learning for self-driving cars." In: *arXiv preprint arXiv:1604.07316* (2016).

[4] Y. Lan Boureau, Jean Ponce, and Yann Lecun. "A theoretical analysis of feature pooling in visual recognition." English (US). In: ICML 2010 - Proceedings, 27th International Conference on Machine Learning. 2010, pp. 111–118.

[5] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN: 9780412048418.

[6] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren D. Cofer. "Input Prioritization for Testing Neural Networks." In: *CoRR* abs/1901.03768 (2019).

[7] Nicholas Carlini and David Wagner. "Towards evaluating the robustness of neural networks." In: *2017 ieee symposium on security and privacy (sp)*. Ieee. 2017, pp. 39–57.

[8] Lidia Ceriani and Paolo Verme. "The origins of the Gini index: extracts from Variabilità e Mutabilità (1912) by Corrado Gini." In: *The Journal of Economic Inequality* 10.3 (2012), pp. 421–443.

[9] Edward Y. Chang. *Knowledge-Guided Data-Centric AI in Healthcare: Progress, Shortcomings, and Future Directions*. 2023. arXiv: 2212.13591 [cs.AI].

[10] Jinyin Chen, Jie Ge, and Haibin Zheng. "ActGraph: Prioritization of Test Cases Based on Deep Neural Network Activation Graph." In: *arXiv preprint arXiv:2211.00273* (2022).

[11] Jinyin Chen, Haibin Zheng, Wenchang Shangguan, Liangying Liu, and Shouling Ji. "ACT-Detector: Adaptive channel transformation-based light-weighted detector for adversarial attacks." In: *Information Sciences* 564 (2021), pp. 163–192.

[12] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system." In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[13] Peng Cui and Jinjia Wang. "Out-of-distribution (OOD) detection based on deep learning: A review." In: *Electronics* 11.21 (2022), p. 3500.

[14] "DeepBackground: Metamorphic testing for Deep-Learning-driven image recognition systems accompanied by Background-Relevance." In: *Information and Software Technology* 140 (2021), p. 106701.

[15] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]." In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[16] Tribikram Dhar, Nilanjan Dey, Surekha Borra, and R Simon Sherratt. "Challenges of Deep Learning in Medical Image Analysis—Improving Explainability and Trust." In: *IEEE Transactions on Technology and Society* 4.1 (2023), pp. 68–75.

[17] Philip M Dixon, Jacob Weiner, Thomas Mitchell-Olds, and Robert Woodley. "Bootstrapping the Gini coefficient of inequality." In: *Ecology* 68.5 (1987), pp. 1548–1551.

[18] Xiangsheng Dou. "Big data and smart aviation information management system." In: *Cogent Business & Management* 7.1 (2020). Ed. by Albert W. K. Tan, p. 1766736. DOI: 10.1080/23311975.2020.1766736.

[19] Bo Du, Zengmao Wang, Lefei Zhang, Liangpei Zhang, Wei Liu, Jialie Shen, and Dacheng Tao. "Exploring Representativeness and Informativeness for Active Learning." In: *IEEE Transactions on Cybernetics* 47.1 (2017), pp. 14–26. DOI: 10.1109/TCYB.2015.2496974.

[20] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. 2022. arXiv: 2109.14545 [cs.LG].

[21] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. "Prioritizing test cases for regression testing." In: *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. 2000, pp. 102–112.

[22] Yang Feng, Qingkai Shi, X. Gao, Jun Wan, Chunrong Fang, and Zhenyu 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* (2020).

[23] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." In: *CoRR* abs/1506.02142 (2015).

[24] Yarin Gal and Zoubin Ghahramani. "Dropout as a bayesian approximation: Representing model uncertainty in deep learning." In: *international conference on machine learning*. PMLR. 2016, pp. 1050–1059.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." In: *arXiv preprint arXiv:1412.6572* (2014).

[26] Ge Han, Zheng Li, Peng Tang, Chengyu Hu, and Shanqing Guo. "FuzzGAN: A Generation-Based Fuzzing Framework for Testing Deep Neural Networks." In: *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE. 2022, pp. 1601–1608.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[28] Thomas A. Henzinger, Anna Lukina, and Christian Schilling. "Outside the Box: Abstraction-Based Monitoring of Neural Networks." In: *CoRR* abs/1911.09032 (2019).

[29] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun 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.

[30] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. "DeepCrime: from Real Faults to Mutation Testing Tool for Deep Learning." In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2023, pp. 68–72.

[31] Alex Kendall and Yarin Gal. "What uncertainties do we need in bayesian deep learning for computer vision?" In: *Advances in neural information processing systems* 30 (2017).

[32] Jinhan Kim, Robert Feldt, and Shin 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.

[33] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." In: *arXiv preprint arXiv:1609.02907* (2016).

[34] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images." In: *University of Toronto* (2009).

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems* 25 (2012).

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[37] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. "Backpropagation applied to handwritten zip code recognition." In: *Neural computation* 1.4 (1989), pp. 541–551.

[38] Ying Li, Zhijie Zhao, Jiahao Fan, and Wenyue Li. "ADR-MVSNet: A Novel Cascade Network for 3D Point Cloud Reconstruction with Pixel Occlusion." In: *Pattern Recognition* (2022), p. 108516.

[39] Yu Li, Min Li, Qiuxia Lai, Yannan Liu, and Qiang Xu. "TestRank: Bringing Order into Unlabeled Test Instances for Deep Learning Tasks." In: *Advances in Neural Information Processing Systems* 34 (2021).

[40] Anna Lukina, Christian Schilling, and Thomas A Henzinger. "Into the unknown: Active monitoring of neural networks." In: *International Conference on Runtime Verification*. Springer. 2021, pp. 42–61.

[41] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions." In: *Advances in neural information processing systems* 30 (2017).

[42] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. "Deepmutation: Mutation testing of deep learning systems." In: *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*. IEEE. 2018, pp. 100–111.

[43] Wei Ma, Mike Papadakis, Anestis Tsakmalis, Maxime Cordy, and Yves Le Traon. "Test selection for deep learning systems." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.2 (2021), pp. 1–22.

[44] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. "Towards deep learning models resistant to adversarial attacks." In: *arXiv preprint arXiv:1706.06083* (2017).

[45] Jean-Baptiste Mouret and Jeff Clune. "Illuminating search spaces by mapping elites." In: *arXiv preprint arXiv:1504.04909* (2015).

[46] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. "Reading digits in natural images with unsupervised feature learning." In: *NIPS* (2011).

[47] Milos Ojdanic, Ahmed Khanfir, Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. "On Comparing Mutation Testing Tools through Learning-based Mutant Selection." In: *4th ACM/IEEE International Conference on Automation of Software Test (AST 2023)*. 2023.

[48] Baida Ouafae, Louzar Oumaima, Ramdi Mariam, and Lyhyaoui Abdelouahid. "Novelty detection review state of art and discussion of new innovations in the main application domains." In: *2020 1st International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*. IEEE. 2020, pp. 1–7.

[49] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. "Deep learning for anomaly detection: A review." In: *ACM computing surveys (CSUR)* 54.2 (2021), pp. 1–38.

[50] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. "The limitations of deep learning in adversarial settings." In: *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE. 2016, pp. 372–387.

[51] Leo Hyun Park, Soochang Chung, Jaeuk Kim, and Taekyoung Kwon. "GradFuzz: Fuzzing deep neural networks with gradient vector coverage for adversarial examples." In: *Neurocomputing* 522 (2023), pp. 165–180.

[52] Roberta Pastorino, Corrado De Vito, Giuseppe Migliara, Katrin Glocker, Ilona Binenbaum, Walter Ricciardi, and Stefania Boccia. "Benefits and challenges of Big Data in healthcare: an overview of the European initiatives." In: *The European Journal of Public Health* 29 (2019), pp. 23–27.

[53] Remigijus Paulavičius and Julius Žilinskas. "Analysis of different norms and corresponding Lipschitz constants for global optimization." In: *Ūkio technologinis ir ekonominis vystymas* 12.4 (2006), pp. 301–306.

[54] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. "Deepxplore: Automated whitebox testing of deep learning systems." In: *proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 1–18.

[55] L Plata-Pérez, Joss Sanchez-Perez, and F Sánchez-Sánchez. "An elementary characterization of the Gini index." In: *Mathematical Social Sciences* 74 (2015), pp. 79–83.

[56] R Pradeepa and K VimalDevi. "Effectiveness of testcase prioritization using apfd metric: Survey." In: *International Conference on Research Trends in Computer Technologies (ICRTCT—2013). Proceedings published in International Journal of Computer Applications®(IJCA)*. 2013, pp. 0975–8887.

[57] Hamzah Al-Qadasi, Yliès Falcone, and Saddek Bensalem. "Difficulty and Severity-Oriented Metrics for Test Prioritization in Deep Learning Systems." In: *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*. In press. IEEE. 2023.

[58] Hamzah Al-Qadasi, Changshun Wu, Yliès Falcone, and Saddek Bensalem. "DeepAbstraction: 2-Level Prioritization for Unlabeled Test Inputs in Deep Neural Networks." In: *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE. 2022, pp. 64–71.

[59] J. Ross Quinlan. "Induction of decision trees." In: *Machine learning* 1.1 (1986), pp. 81–106.

[60] Laura Elena Raileanu and Kilian Stoffel. "Theoretical comparison between the gini index and information gain criteria." In: *Annals of Mathematics and Artificial Intelligence* 41.1 (2004), pp. 77–93.

[61] Waseem Rawat and Zenghui Wang. "Deep convolutional neural networks for image classification: A comprehensive review." In: *Neural computation* 29.9 (2017), pp. 2352–2449.

[62] Faqeer Ur Rehman and Madhusudan Srinivasan. "Metamorphic Testing For Machine Learning: Applicability, Challenges, and Research Opportunities." In: *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE. 2023, pp. 34–39.

[63] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks." In: *Advances in neural information processing systems* 28 (2015).

[64] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "" Why should i trust you?" Explaining the predictions of any classifier." In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[65] Michael D Richard and Richard P Lippmann. "Neural network classifiers estimate Bayesian a posteriori probabilities." In: *Neural computation* 3.4 (1991), pp. 461–483.

[66] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer. 2015, pp. 234–241.

[67] Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. ""Everyone Wants to Do the Model Work, Not the Data Work": Data Cascades in High-Stakes AI." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445518.

[68] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. "Grad-cam: Visual explanations from deep networks via gradient-based localization." In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 618–626.

[69] C. E. Shannon. "A mathematical theory of communication." In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[70] Y-S Shih. "Families of splitting criteria for classification trees." In: *Statistics and Computing* 9.4 (1999), pp. 309–315.

[71] Ann Sizemore, Chad Giusti, and Danielle S Bassett. "Classification of weighted networks through mesoscale homological features." In: *Journal of Complex Networks* 5.2 (2017), pp. 245–273.

[72] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[73] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic attribution for deep networks." In: *International conference on machine learning*. PMLR. 2017, pp. 3319–3328.

[74] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going Deeper with Convolutions." In: *CoRR* abs/1409.4842 (2014).

[75] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In: *International Conference on Machine Learning (ICML)*. 2019.

[76] Chuanqi Tao, Yali Tao, Hongjing Guo, Zhiqiu Huang, and Xiaobing Sun. "DLRegion: Coverage-guided fuzz testing of deep neural networks with region-based neuron selection strategies." In: *Information and Software Technology* (2023), p. 107266.

[77] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. "Deeptest: Automated testing of deep-neural-network-driven autonomous cars." In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 303–314.

[78] *Understanding the fatal Tesla accident on Autopilot and the NHTSA probe*. Accessed: 2018-08-29. 2016. URL: https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/.

[79] Unknown. *A Google self-driving car caused a crash for the first time*. Accessed: 2018-08-29. 2016. URL: http://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report.

[80] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11 (2008).

[81] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. "Deep learning library testing via effective model generation." In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 788–799.

[82] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. "Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 397–409.

[83] Changshun Wu, Yliès Falcone, and Saddek Bensalem. "Customizable Reference Runtime Monitoring of Neural Networks using Resolution Boxes." In: *CoRR* abs/2104.14435 (2021).

[84] Mingfei Wu, Chen Li, and Zehuan Yao. "Deep Active Learning for Computer Vision Tasks: Methodologies, Applications, and Challenges." In: *Applied Sciences* 12.16 (2022), p. 8103.

[85] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms." In: *CoRR* abs/1708.07747 (2017).

[86] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. "Deephunter: a coverage-guided fuzz testing framework for deep neural networks." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 146–157.

[87] Rongjie Yan, Yuhang Chen, Hongyu Gao, and Jun Yan. "Test case prioritization with neuron valuation based pattern." In: *Science of Computer Programming* 215 (2022), p. 102761.

[88] Kai Zhang, Yongtai Zhang, Liwei Zhang, Hongyu Gao, Rongjie Yan, and Jun Yan. "Neuron Activation Frequency Based Test Case Prioritization." In: *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2020, pp. 81–88. DOI: 10.1109/TASE49443.2020.00020.

[89] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 132–142.

[90] Alice Zheng and Amanda Casari. *Feature Engineering for Machine Learning*. O'Reilly Media, Inc., 2018. ISBN: 9781491953235. URL: https://learning.oreilly.com/library/view/feature-engineering-for/9781491953235/.

[91] Haibin Zheng, Jinyin Chen, Hang Du, Weipeng Zhu, Shouling Ji, and Xuhong Zhang. "GRIP-GAN: An attack-free defense through general robust inverse perturbation." In: *IEEE Transactions on Dependable and Secure Computing* 19.6 (2021), pp. 4204–4224.

[92] Haibin Zheng, Jinyin Chen, and Haibo Jin. "CertPri: Certifiable Prioritization for Deep Neural Networks via Movement Cost in Feature Space." In: *arXiv preprint arXiv:2307.09375* (2023).

[93] Jinyi Zhou, Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. "Using Metamorphic Testing to Evaluate DNN Coverage Criteria." In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2020, pp. 147–148.

[94] Zhi Quan Zhou, Junting Zhu, Tsong Yueh Chen, and Dave Towey. "In-place metamorphic testing and exploration." In: *Proceedings of the 7th International Workshop on Metamorphic Testing*. 2022, pp. 1–6.

[95] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. "Deephyperion: exploring the feature space of deep learning-based systems through illumination search." In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 79–90.

[96] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. "Efficient and effective feature space exploration for testing deep learning systems." In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023), pp. 1–38.