THE UNIVERSITY *of* EDINBURGH

Testing Deep Neural Networks Across Different Computational Configurations

Nikolaos Louloudakis

Doctor of Philosophy

The University of Edinburgh

2025

Abstract

Deep Neural Networks (DNNs) typically consist of complex architectures and require enormous processing power. Consequently, developers and researchers use Deep Learning (DL) frameworks to build them (e.g., Keras and PyTorch), apply compiler optimizations to improve their inference time performance (e.g., constant folding and operator fusion), and deploy them on hardware accelerators to parallelize their computations (e.g., GPUs and TPUs). We[1] concisely refer to these aspects as the *computational environment* of Deep Neural Networks.

However, the extent to which the behavior of a DNN model (i.e., output label inference correctness and computation times) is affected when different configurations are selected across the computational environment, is overlooked in the literature. For example, if a DNN model is deployed on two different GPU devices, will it give the same predictions, and how will its computation times deviate across the devices? Given that DNNs are deployed on safety-critical domains (e.g., autonomous driving), it is important to understand the extent to which DNNs are affected by these aspects.

For that purpose, we present `DeltaNN`, a tool that allows DNN model compilation and deployment under different configurations, as well as comparison of model behavior across them. Using `DeltaNN`, we conducted a set of experiments on widely used Convolutional Neural Network (CNN) models performing image classification[2]. We built these models using different DL frameworks, converted them across different DL framework configurations, compiled on a set of optimizations and deployed on GPU devices of varying capabilities. Our experiments with different configurations led to two main observations: (1) while DNNs typically generate the same predictions across different GPU devices and compiler optimization settings, this is not true when utilizing different DL frameworks, and especially when converting from one DL framework to another (e.g., converting from Keras to PyTorch), a common practice across developers to enable model portability and extensibility; and (2) optimizations are not a panacea of inference time improvement across different devices, as the same optimization strategies that improve execution times on high-end GPUs were found to degrade them when applied on models deployed on low-end GPUs.

[1] Although this is my thesis, I sincerely believe this is a contribution that I worked along with my supervisors and fellow collaborators throughout the years of my doctorate studies. This way, I recognize their support. All publications related to this thesis, rightfully list them as co-authors.

[2] However, this methodology is applicable to other model architectures as well.

To mitigate the faults related to the conversion process, we implemented a framework called FetaFix. FetaFix performs automatic fault detection by comparing a number of aspects across the source and the converted target DNN model, such as model parameters, hyperparameters and structure. It then applies a number of fault repair strategies related to these aspects and checks how the converted model performs in comparison to its source counterpart. FetaFix was able to repair 93% of the problematic cases identified by DeltaNN.

Finally, we explored the effects of faults present in the target hardware acceleration device code towards DNN model correctness. Inspired by traditional mutation testing, we built MutateNN, a tool that generates DNN model mutants containing target device code faults. We then generated a number of faults in the target device code of numerous CNN models performing classification and evaluated how these models behaved across different hardware acceleration devices. We observed that faults related to conditional operations, as well as drastic changes in arithmetic types, considerably affected model correctness.

We conclude that different configurations of computational environment aspects can affect DNN model behavior. Our contributions summarize to (1) an empirical study on how the computational environment affects DNN model behavior, performed by a tool (DeltaNN) implemented specifically for that purpose, (2) a framework (FetaFix) that automatically detects faults related to model input, structure and parameters in converted DNN models across DL frameworks and repairs them, and (3) a utility (MutateNN) that introduces faults in the target code of DNN models associated with deployment on different hardware acceleration devices, and evaluates the effects of these faults on model correctness.

Acknowledgements

First of all, I would like to wholeheartedly thank my supervisors, Prof. Dr. Ajitha Rajan and Prof. Dr. José Cano Reyes, for their continuous guidance and support throughout my PhD studies. I could not have asked for better supervisors, and this is no exaggeration. You made my Doctorate studies an enjoyable learning and development journey, that transformed me into a better researcher, professional and person. You showed empathy in difficult times, motivated me and never gave up on me. For that, I am, and will be, Always Grateful.

In an equally important manner, I would like to thank my family for their continuous and unstoppable support in this Ph.D. marathon. My father Konstantinos, my mother Andriani and my brother Stefanos, I am blessed with your unconditional, continuous love, support and encouragement in times I felt I could not make it any longer, but also with your faith in me. Without you, I would not have completed this journey. I Love You And You Mean *The World* To Me.

In addition, I would like to thank my colleagues and fellow PhD students for their ideas, support, and collaboration, both in the University of Edinburgh and the University of Glasgow. Special thanks to Perry Gibson, Chao Peng and Foivos Tsimpourlas. Our discussions really helped me, and your insights opened the road many times in my Ph.D. quest.

I am also grateful for the support that I received from my lifelong friends. I would like to give special thanks to Chris Melessanakis, Bart Papadokonstantakis, Nikolas Siatos, Yiorgos Stathopoulos, Vasileios Theodosiadis, John Tzortzakis and Nick Vervelakis, for your amazing support and motivation all these years. Our friendship made me a better person, in the same way that iron sharpens iron, and helped me finish this good fight.

Last but not least, in the manner of the Senior Author who guides and supports the journey, I would like to thank and give praise to my personal Lord and Savior, Jesus Christ. I truly believe that he has guided my steps through this beautiful adventure, in both good times and challenging ones, and that He is with me always.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Nikolaos Louloudakis)

Dedicated To My Family.

Be Strong & Courageous...

Contents

Figures and Tables

Figures

Tables

Chapter 1
Introduction

1.1 Overview

In the present day, Artificial Intelligence (AI) is emerging tremendously. The domain of Machine Learning (ML) has rapidly evolved in the last decade, allowing humanity to achieve previously unachievable solutions and advances in a variety of demanding and challenging domains, such as the automotive industry [2], medical applications [3], space exploration [4] and software engineering [5]. Deep Neural Networks (DNNs) are utilized in order to enable technological advances such as autonomous driving, discovery of disease treatments and more. However, such domains require that AI performs in an efficient and robust manner, as a compromise to their behavior could lead to catastrophic consequences, both financially, but most importantly, at the cost of human life [6]. Consecutively, these AI systems need to be extensively tested, and the scientific community has proposed a number of methodologies concerning DNN model training and execution. For the training phase, some examples include the usage of adversarial, "poisonous" data [7, 8, 9, 10], mutation testing [11, 12] and partial neuron freezing and retraining [13]. For the execution phase, examples of techniques include but are not limited attacks consisting of adversarial inputs [14], fault injection such as bit flip attacks, instruction skipping, weight modification and data corruption [15], and mutation testing and input validation involving metamorphic relations [16]. However, various aspects related to implementation, optimization and deployment of DNNs have been overlooked by the literature, as there are limited works towards the testing of Deep Learning (DL) frameworks (e.g., [17, 18]), model optimizations (e.g., [19, 20]) and hardware acceleration (e.g., [21, 22]).

This thesis aims to contribute to this direction, by (1) presenting a tool for DNN model testing across the aforementioned parameters and configurations, (2) proposing a framework for automatic fault localization and repair for erroneous DNN model conversions across DL frameworks (a process widely utilized by DNN developers

for portability and compatibility purposes), and (3) explore the effects of device code miscompilations to DNN model behavior (i.e., compiler faults that produce code violating the specification of the compiler and, consequently, cause unintended behavior) and robustness when deployed on different hardware accelerators.

Given that all the aspects we explore (DL frameworks, compiler optimizations, deployment on hardware accelerators) contribute vitally to the computational correctness and inference (execution) time performance of DNNs, we typically refer to them as the **Computational Environment** of DNNs. We present in depth our contributions in the following sections.

1.2 Software Testing of Deep Neural Networks

Software testing is a vital part of the software development process to verify that software systems work the way they are intended, and to reduce development and maintenance costs, with numerous techniques having evolved throughout the years for that purpose, such as unit and integration testing [23]. On the other hand, inadequate testing can have catastrophic financial (e.g., Ariane 5 Flight [24]) and life-threatening (e.g., Therac-25 [25]) consequences.

Similar to software systems, DNNs also must be tested to ensure they work as intended. However, DNNs consist of complex architectures and their code does not define the control flow, but their business logic is formed by large amounts of data and iterative training processes [26]. This results in DNNs being perceived and used as "black boxes", difficult to interpret and understand. In turn, this leads to challenges in their testing [27], proving techniques used in traditional software being ineffective without the necessary adaptation [28]. In addition, DNNs are utilized in a wide variety of applications, quite commonly in the safety-critical domain (e.g., autonomous vehicles), where a system error can result in catastrophic consequences affecting human life. As a result, such systems must adhere to much stricter standards compared to non-safety-critical systems, where errors are more tolerable and can often be mitigated. However, the definition of standards that involve DNNs in safety-critical domains, are still under development and face considerable challenges [29, 30]. ISO 26262 [31, 32] addresses functional safety of electronic systems in production vehicles, however lacks consideration of DNNs, unless they are part of a specific safety mechanism. ISO/PAS 21448 (also known as a standard related to Safety of the Intended Functionality - SOTIF [33, 34] is a standard that safety risks arising from the performance limitations and the unintended behavior

occuring in correctly functioning systems, which includes DNN-based perception and decision-making systems. However, it lacks detailed technical guidance for challenges related to DNNs, such as explainability, adversarial robustness, uncertainty estimation, and continuous learning, and is not compatible with fully autonomous vehicles (SAE Level 5, according to the SAE J3016 standard [35], which defines levels of driving automation). ISO/IEC TR 24029 [36] provides guidance on evaluating the robustness and uncertainty of AI systems, including DNNs, to support their trustworthiness in safety-critical applications. However, although it offers techniques to achieve such aims, it is not tailored to real-life applications and does not provide a way to integrate them with broader safety standards. UL 4600 [37] is a proprietary safety standard for the evaluation of autonomous products, taking DNNs into consideration, first released in 2020 and still evolving. However, it does not fully address AI-specific challenges like adversarial robustness and model explainability in safety-critical systems. IEEE P2851 [38] focuses on the safe implementation of AI systems, including DNNs, in autonomous vehicles. It provides guidelines for their operation, validation, and risk management. However, it is still under development, and offers general guidance rather than specific technical requirements, which limits its direct applicability in real-world applications.

Studies have shown that many of these DNN systems suffer from flaws and vulnerabilities, and that their efficient usage can be challenging [39] - especially in safety-critical domains, with the need of establishing testing procedures becoming vital [40].

The scientific community has attempted to identify and progress DNN testing, with efforts to explore and categorize the causes of DNN faults [41, 42], along with a considerable number of testing methodologies suggested [43], such as adversarial testing [7, 14, 44] and fuzzing [16, 45] adapted to DNNs. However, the effects and testing of numerous aspects associated with their implementation, optimization and deployment have been overlooked by the literature.

1.3 The Computational Environment in Deep Neural Networks

Similar to traditional software, DNNs are designed, built and deployed by utilizing a number of different parameters and factors that vitally contribute to their overall performance in the directions of inference correctness and execution times. In particular, DNNs are implemented and operate on top of DL libraries and frameworks that provide mature and well-tested implementation of Application Programming Interfaces (APIs) for all the building blocks and algorithms necessary to build, train and deploy DNNs. In addition, following DNN model compilation, a variety of optimizations are applied in order to improve their performance. Finally, as DNN models typically consist of multi-dimensional vector computations, they are commonly deployed on hardware acceleration devices in order to speed up DNN computations. We refer to these aspects as the *Computational Environment*. An overview of the DL systems stack can be found at Figure 1.1.

In order for DNNs to work optimally, it is expected that none of the computational environment aspects contribute negatively towards model correctness and performance. However, there is a plethora of different yet complex implementations of DL frameworks (e.g., PyTorch [46] and Keras [47]) as well as DNN model optimization algorithms [48], each following their own design decisions, implementation business logic and conventions. aIn addition, hardware acceleration devices may follow different architectures and rely on different programming frameworks for hardware acceleration computations (e.g., OpenCL [49] and CUDA [50]).

Furthermore, given the complexity of DNNs, as well as the challenges related to each of these aspects, configurations that use different combinations of them can potentially affect a model both in terms of correctness of prediction accuracy and computational efficiency (i.e., inference computation times).

In addition, DNN model robustness needs to be examined against these parameters, but most of these aspects have been overlooked in the literature. We aim to explore the effects that changes in any of these aspects have in model behavior. As one of the primary contributions in this thesis, we propose a mechanism (called *DeltaNN*) in order to explore the effects of the computational environment on DNN models upon deployment, focusing on two performance aspects: (1) model correctness (i.e., prediction accuracy) and (2) inference times (i.e., execution times to perform inference).

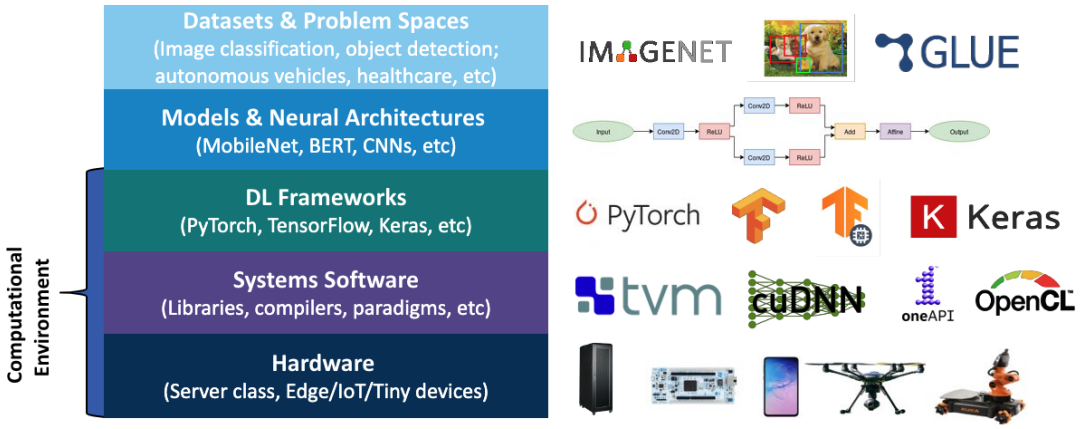


Figure 1.1: The Deep Learning systems stack. We identify the bottom three layers as the computational environment of DNNs.

1.3.1 Deep Learning Frameworks

DNNs are built, trained, optimized and deployed using a widely available variety of DL libraries and frameworks, such as Keras [47], TensorFlow [51] and PyTorch [46]. Such frameworks provide APIs for the development of DNNs, including well-established algorithm implementations for common DNN constructs and operations, such as numerous activation functions (e.g., ReLU), and tensor computations (e.g., convolutions). In addition, some frameworks focus on specific tasks, such as enabling the compilation of models for edge device deployment (e.g., TFLite [51]), as well as the ability of extensive optimizations for hardware acceleration (e.g., PyTorch [46]). However, there is no standard way for implementing DNN constructs and features in DL frameworks, with the developers of each framework taking their own design decisions. For example, PyTorch and Keras handle padding differently in a convolutional layer [52]. Furthermore, testing that DNN models behave consistently across DL frameworks is crucial.

Deep Learning Framework Conversions

A common practice that developers and researchers follow, is the conversion of already trained DNN models across different DNN frameworks. This is widely applied for two main reasons: (1) to enable model portability, by converting it to a DL framework targeting edge devices (e.g., from PyTorch to TFLite), and (2) model extensibility, such as converting a model to a framework that provides a lower-level GPU optimization API in comparison to the DL framework that the model was

initially built on (e.g., Keras as source, converted to PyTorch). Given the large cost and complexity of retraining models in the target frameworks, developers quite commonly choose to convert the models as-is. For that purpose, numerous tools have been implemented and widely used [53, 54, 55, 56, 57], typically utilizing industry-wide established standards, such as Open Neural Network Exchange (ONNX) [58]. However, given the plethora of converters and the complexity of the systems they operate on, the reliability of the conversion process should be examined.

1.3.2 Compiler Optimizations

DNNs typically consist of layers that handle multi-dimensional arrays that are used to describe physical properties in data - called tensors. For the purpose of optimizing the performance of these computations, various optimization techniques have been implemented and utilized upon model compilation [48]. Techniques such as constant folding and axis scaling, apply graph-level modifications to the DNN model with the purpose of optimizing its inference time performance. However, these techniques usually come with trade-offs that might affect model correctness, and their usage must be further examined. For example, fast-math optimization allows the model to not strictly adhere to IEEE-754 floating point arithmetic specification [59], enabling faster computation, with the risk of potentially giving imprecise results at times.

1.3.3 Hardware Acceleration Devices

As aforementioned, DNNs consist of layers of tensor operations. Since such operations are correlated with large amounts of computations that can be parallelized, it is typical that DNNs are deployed on hardware acceleration devices such as GPUs and TPUs, to accelerate their performance. However, the deployment in such devices is not straightforward, as a variety of architectures exist. In addition, the code orchestrating a hardware accelerator is non-straightforward and error-prone, with considerable learning curve for developers to write efficient hardware acceleration code. As an attempt to mitigate this, AI/ML compilers and frameworks have been implemented [60, 61]. However, compilers are inherently complex systems, and their correctness must be thoroughly validated. This applies equally to AI/ML compilers. In the past, considerable research has been devoted to this challenge across various domains and types of compilers (e.g., graphic shaders [62, 63], hardware acceleration [64], and generic-purpose programming languages, such as `C` [65]).

1.4 Motivation

This thesis aims to contribute to the direction of exploring the aforementioned computational environment aspects of DNNs: (1) DL frameworks (including conversions), (2) compiler optimizations, and (3) deployment across different hardware acceleration devices. Our motivation comes from the facts that (1) there is a lack of testing standards for DNNs, (2) these aspects play a crucial role to DNN implementation, deployment, optimization and computation, (3) each of these aspects contains its own technological complexity that adds up to the already sophisticated nature of DNNs, (4) there is a constant evolution and increase in the variety of technologies related to all of these aspects, and that (5) exploring their effects in DNN correctness and performance is widely overlooked by the literature.

As an example, imagine that a DNN model architecture is defined, e.g., ResNet101 [66, 67] - a model used for classification and object detection tasks. This model can be implemented using a plethora of DL frameworks, with each one providing APIs that should behave similarly but contain differences in their implementations. Imagine now that a developer wants to convert an already trained ResNet101 model, from Keras [47] (a high-level DL frameworks) to PyTorch [46] (a DL framework providing richer API for hardware acceleration). By doing so, the developer would expect the two models to behave the same, given that their architecture has not changed and no further training is applied. However, there is no standard guarantee for this, no matter how intuitive it might seem. In fact, for this particular example, our experiments indicated 92% discrepancies for models trained in Imagenet [68] and tested across the ILSVRC 2017 [69] dataset. We present our contributions in this thesis.

1.5 Contributions

This thesis makes the following main contributions:

- `DeltaNN` [70], a tool that enables differential testing (i.e., execution of different system configuration against the same dataset and comparison of the results) for image recognition AI models across different computational environment configurations. It is accompanied by a full set of experiments and a comprehensive study on the effects of the computational environment on DNN model robustness on model output correctness and execution time performance. The

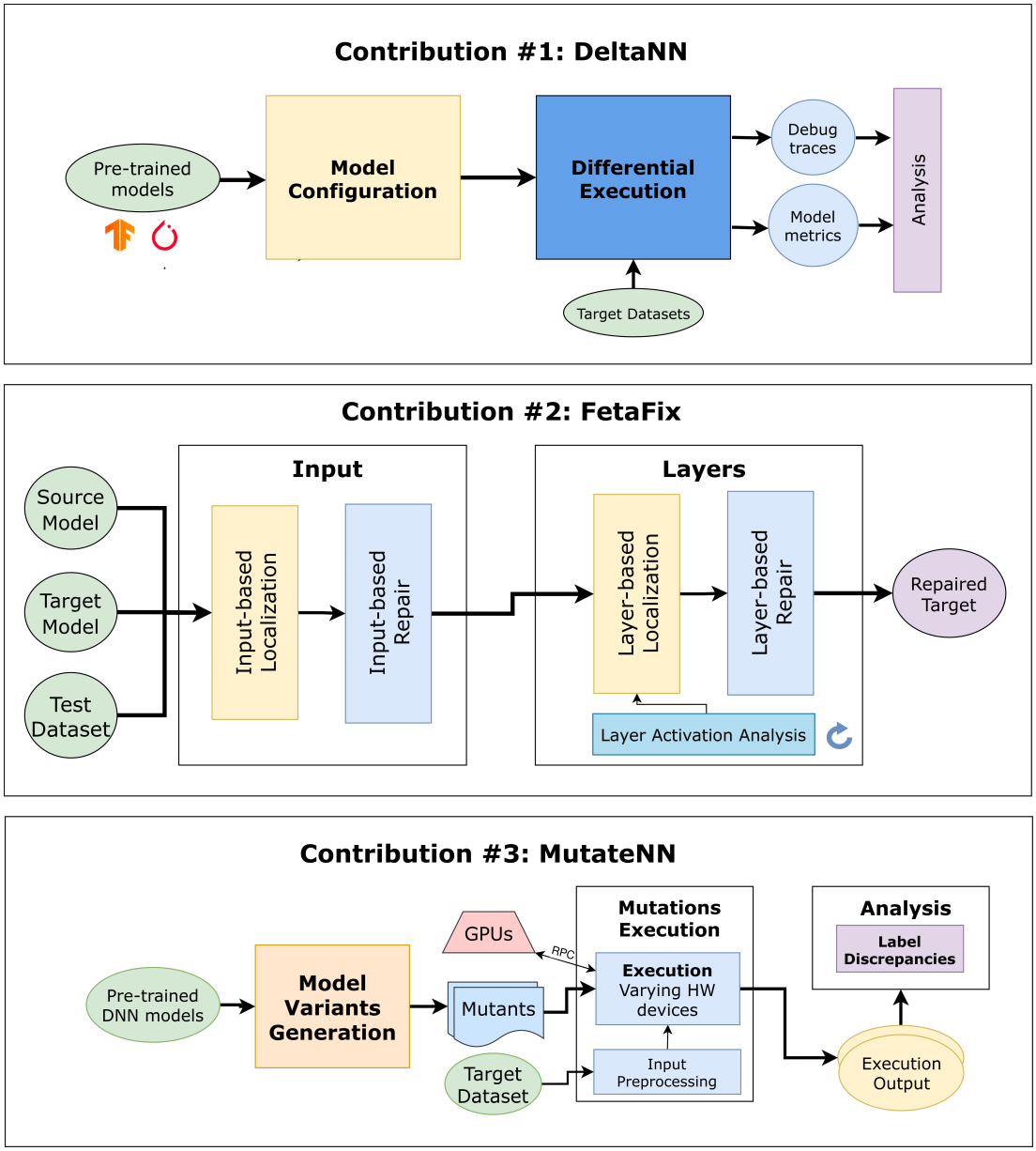


Figure 1.2: Overview of Thesis Contributions.

study contains a comprehensive set of experiments in well-known models, across different configurations concerning DL frameworks and their conversions, compiler optimizations and performance (in terms of both accuracy and execution times) under different hardware acceleration devices.

- `FetaFix`, a framework for automatic fault localization and repair of DNN models affected by problematic DL framework conversions - a computational environment parameter that was shown to heavily affect model correctness. `FetaFix` utilizes statistical analysis in order to detect model layers mostly affected by the problematic conversion process. It then applies a variety of input and layer-based repair strategies, inspired by conversion issues sourced from codebase issue reports, Q&A websites, and forums.

- `MutateNN`, a tool that explores the effects of problematic target code on image recognition models when deployed on different hardware accelerators. `MutateNN` enables mutant generation on the low and high-level intermediate representations (IRs) of Apache TVM [60], an open-source and well-established AI compiler framework that enables compilation of target code for a wide variety of hardware acceleration devices.

A visual depiction of these contributions can be found in Figure 1.2. We present an overview of each work in the following introductory sections. A detailed description of each contribution can be found in the next chapters.

1.5.1 DeltaNN: Differential Testing of AI Against the Computational Environment

For the purposes of research tractability, we choose to explore the impact of various computational environment aspects of perception AI models performing image classification. However, our approach is generic and can be extended to DNN models performing object detection and semantic segmentation, as well as models of different architectural designs, such as Recurrent Neural Networks (RNNs) and transformers.

Our work is summarized under a tool that explores DNN model behavior under different computational environment aspects, named *DeltaNN*. `DeltaNN` allows: (1) compilation of DNN models using different DL frameworks, (2) automated model conversion across a plethora of DL frameworks, (3) application of different optimizations to the models, and (4) model deployment on different hardware acceleration devices. Finally, `DeltaNN` allows the comparison of execution results against all model configurations.

Using DeltaNN, we conducted a series of experiments, considering two metrics regarding model performance: (1) output label prediction correctness (comparing top-1 label output classification predictions), and (2) inference (execution) times. Overall, we utilized 3 well-known image recognition models utilized for classification tasks (MobileNetV2 [71], ResNet101 [66, 67] and InceptionV3 [72]), built under 4 popular, well-known DL frameworks (Keras [47], PyTorch [46], TensorFlow(TF) [51], and TF Lite [51]), while also generating all (36) potential combinations of DL framework conversions, which is as a result of all permutations without replacement across the DL frameworks under test for each model, multiplied by the number of models $(12 \times 3 = 36)$. We also applied 3 optimization batch settings across each model (no optimizations, typical graph-based optimizations and heavy optimizations), and deployed them in 4 hardware acceleration devices of varying capabilities. All experiments were performed using a well known validation dataset [69], consisting of 5500 images.

The results indicated that DL frameworks affected model behavior, as a DNN model built with different DL frameworks performed different top-1 label predictions. In particular, we observed up to 49% top-1 label prediction differences (on MobileNetV2, when comparing the model variants built using Keras and PyTorch). Most importantly, model conversion across DL frameworks [1], was observed to be heavily prone to errors. We elaborate further on this observation in Section 1.5.2.

In addition, the results indicated that (1) the DNN models were found to be robust in terms of output label correctness across different hardware acceleration devices, and (2), optimization effectiveness can be dependent on hardware acceleration capabilities, as we observed optimizations that increased execution time in high-end devices, to lead in severe execution time degradation in low-end devices.

1.5.2 FetaFix: Fault Localization and Repair of Buggy Deep Learning Framework Conversions

Our experiments indicated that DL framework conversions can introduce errors to DNNs - a common practice across developers and researchers, for reasons of model portability, extensibility and optimization. Upon conducting the experiments with DeltaNN, we observed that, when models were converted from one DL framework to another, a considerable percentage of the conversions either failed, or resulted in problematic

[1] Model conversion across DL frameworks is a process applied commonly in academic research and industry for purposes of model extensibility, optimization and portability.

target models. In particular, 30.5% of the conversions resulted in a crash, but most importantly, 41% of them (15 cases out of 36) resulted in different predictions between the original and the converted model, to the range of 3-100%. We chose to further explore this domain, deeming more important to focus on the cases that correctness degradation was applied. We inferred that problems introduced in the models in these cases might be overlooked and be harder to find, while playing a more significant role in the context of AI model use. Despite the high error rate, model developers might still choose to convert models using converters, since the alternative (rebuilding the model from scratch in the target deep learning framework and retraining it) is costly and computationally intensive.

In order to explore the reasons behind these observations, we sourced 40 errors in relation to the model conversion process out of codebase issue trackers and Q&A websites and forums, and identified 6 potential sources of faults related to the conversion process. Then, for each of these sources, we proposed a fault mitigation strategy. We also constructed a proof of concept for one of the strategies related to precision errors in model weights, in which case we were able to successfully localize and repair the fault. As a next step, we implemented a full fault localization and repair framework for model conversions, called *FetaFix*. `FetaFix` consists of: (1) a statistical analysis module that enables the detection of layers mostly affected by the problematic conversion, utilized for fault localization purposes; and (2) an automatic repair module that employs the automatic repair strategies previously formulated in the converted model. `FetaFix` managed to drastically improve the accuracy of 14 out of the 15 cases, with 12 of them being repaired completely, giving the same results with the model variant used before conversion.

1.5.3 MutateNN: Mutation Testing of Image Recognition Models on Compiler-Related Hardware Acceleration Faults

A recent study [41] indicated that GPU faults (e.g., wrong GPU device reference and incorrect state sharing) are a primary source of errors in DNNs. In addition, compiling target code for DNN model deployment in hardware acceleration devices is a non-trivial and challenging task due to the complexity of hardware acceleration kernels. To this direction, AI/ML compilers have been implemented (e.g., TVM [60]), which also bear the burden of generating the target device code. In addition, frameworks for hardware acceleration programming evolve, (e.g., Triton [73]), allowing developers to implement their own solutions.

In order to investigate the extent of effects of faults present in target code (coming from either compiled code that is malformed or behaves incorrectly (miscompilation) or bugs introduced by developers), we implemented a system named MutateNN. Inspired by traditional mutation testing, MutateNN allows the generation of valid AI model mutants by modifying the model intermediate representations, both in graph (Relay IR) and kernel (TIR) level; thus enabling the execution of each mutant across different devices and evaluating their behavior in the manner of differential testing. Differential testing allows us to compare the correctness of identical configurations across multiple devices and, as a result, effectively assess how faults affect each device's behavior. To evaluate the efficiency of MutateNN, we selected 7 image recognition models and applied 21 mutations out of 6 categories, both graph and target code-related. We then deployed the mutants across 4 hardware acceleration devices and compared the results across them. Our findings indicated that mutations related to conditional expressions could lead to diverse results across hardware acceleration devices, meaning that a fault could have no effects on one device, but cause accuracy deviations on another. In addition, mistakes in types could also severely affect model correctness, but without leading to a model crash - an effect observed across all devices under test. Therefore, developers should test models on multiple devices and apply well-defined thresholds within their testing suites throughout the development process.

As a first approach, with MutateNN, we explore the effects of the faults across the hardware acceleration devices. The localization of these faults is outside the scope of this work.

1.5.4 Contributions Overview

An overview of our contributions can be found in Figure 1.2. In particular, the figure presents 3 systems that contribute towards (1) the differential testing of DNNs in the context of their computational environment (DeltaNN), (2) the fault localization and repair of image recognition models for a process we discovered to be error prone - the DL framework conversions (FetaFix), and (3) the mutation testing of AI compilers in the context of target code generated for hardware accelerators (MutateNN).

1.6 Publications

This thesis consists of six publications (including one preprint of preliminary work), presenting the ideas, methodology and results.

In detail, the `DeltaNN` suite, including its methodology and results, is presented in the following publications:

- "Assessing Robustness of Image Recognition Models to Changes in the Computational Environment", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in NeurIPS ML Safety Workshop 2022 [74].
- "DeltaNN: Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in IEEE ICSME 2023 [70].
- "Exploring Effects of Computational Parameter Changes to Image Recognition Systems", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in ArXiv (preprint of preliminary work) [75].

`FetaFix`, including its concept, implementation and result is presented in comprehension in the following publications:

- "Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in IEEE/ACM ASE 2023 [76].
- "FetaFix: Automatic Fault Localization and Repair of Deep Learning Model Conversions", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in EASE 2025 [77].

The concept and results of `MutateNN`, are presented in:

- "Exploring Robustness of Image Recognition Models on Hardware Accelerators", **N. Louloudakis**, P. Gibson, J. Cano, A. Rajan, in IEEE ICST Mutation 2025 [78].

As first author in these publications, I played a leading role in driving the research efforts. More specifically, I conceptualized the research, designed the methodology, implemented the respective software systems, conducted the necessary experiments, and finally analyzed the results to extract meaningful observations and insights. I made

significant contributions to the writing process of the publications, which consisted of multiple iterations and revisions, following the valuable guidance, feedback and suggestions from my supervisors and collaborators. My contributions also extended towards managing the paper submission process, addressing comments and suggestions provided by reviewers and presenting the works at the respective venues.

1.7 Thesis Structure

This thesis is structured as follows:

- **Chapter 2** presents all the background information and related work relevant to this thesis, organized into subsections.
- **Chapter 3** presents `DeltaNN`, a tool for differential testing of DNNs under different computational environment aspects. The chapter includes (1) information about its methodology and implementation, (2) a comprehensive experiment set and (3) an overview of the results.
- **Chapter 4** presents `FetaFix`, a framework that allows automatic fault localization and repair of DL framework conversions. The chapter contains information about (1) six types of faults identified as potential sources of faults in the conversion process, (2) a methodology that attempts fault localization of such faults, (3) a comprehensive set of experiments against models that were erroneously converted, and (4) a full presentation of results.
- **Chapter 5** presents `MutateNN`, a tool that allows mutation testing for automatically generated code from AI compilers. The chapter presents (1) the tool methodology, and (2) an indicative set of experiments as proof of concept, along with their results.
- **Chapter 6** Summarizes the contributions of this thesis, reflects on the work conducted and outlines potential avenues and directions for future work.
- **Appendix A** contains supplementary material and results related to Chapter 3.

Chapter 2
Background & Related Work

2.1 Overview

This chapter presents the necessary background information for the reader, as well as an overview of the related work of this thesis. We present all necessary terms, accompanied with the existing literature related to our contributions. In particular, we present the existing work conducted across the Deep Learning System Stack [79], as presented in Figure 2.1. Our contributions primarily focus on the three bottom layers of the stack, which we identify as the computational environment of DNNs.

As a preliminary note, we emphasize that:

- DNNs cannot be tested using conventional methods like functional and branch coverage due to their data-driven nature and lack of traditional control-flow. Existing testing approaches need improvement in defining effective coverage criteria, input selection, and test oracle definitions for DNNs [28].
- The definition of DNN coverage criteria in particular, differs significantly from those in conventional software systems. Since DNNs are typically graph-based and their behavior is driven primarily by data rather than explicit code execution paths and do not follow a traditional control-flow structure [26], coverage in this context must be defined using alternative criteria [80].
- Similarly to conventional software, coverage metrics are not adequate by themselves to localize faults, but can be effective when combined with other techniques, such as adversarial testing [45] and fuzzing [81].

- Testing DNN models is a challenging task. Existing contributions have extensively surveyed the literature as well as codebases, issue trackers, forums and domain experts, forming taxonomies [41, 42] with a wide variety of issues, as observed in Section 2.4. Most importantly, the computational environment aspects are identified, and in fact highlighted among the challenging aspects for DNN model development and deployment.

- The generation of inputs designed to degrade model performance or cause crashes is a primary focus of DNN testing literature. This approach, known as *adversarial testing*, constitutes a significant portion of the contributions in the field, as discussed in Section 2.4.2.

- Common real-world faults [41] examined when running DNN models in an offline setup (i.e., upon controlled mode execution and behavior inspection) can prove quite challenging to detect when running DNNs in a real-life production scenario [82]. In addition, the model deployment process can prove challenging, as there is a wide spectrum of faults that can occur, making it difficult for developers to localize and fix them [83].

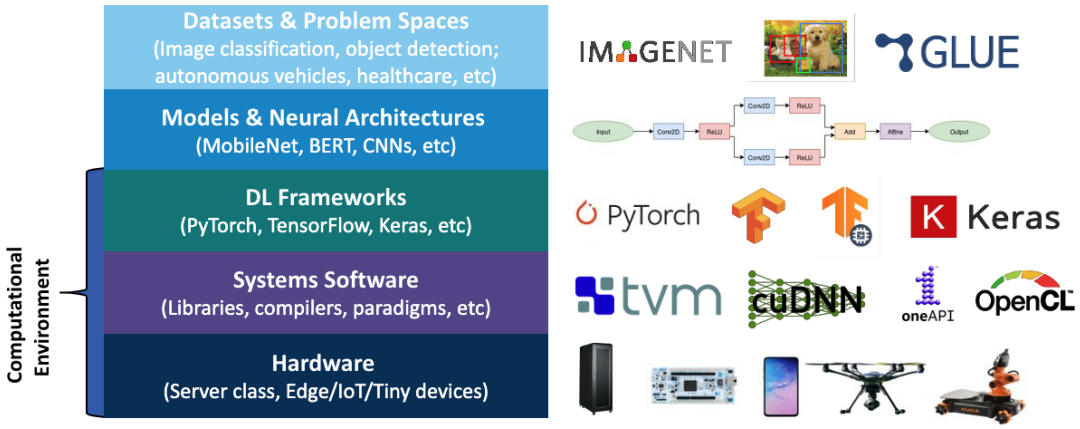


Figure 2.1: The Deep Learning systems stack. We identify the bottom three layers as the computational environment of DNNs (reintroduced from Chapter 1).

2.2 Machine Learning

Machine Learning (ML) is a paradigm concerning software systems (models) whose behavior is shaped by a learning process over usually substantial amounts of data. ML models are used in a variety of tasks such as image recognition and media generation, for purposes such as autonomous driving, medical applications, but also generic-purpose content generation. ML is typically classified in four categories, based on the learning process applied into the system:

1. **Supervised Learning:** ML models learn based on fully labeled data, quite commonly a product of manual classification.
2. **Unsupervised Learning:** ML models learn based on not strictly labeled or clearly structured data by attempting to identify patterns and similarities.
3. **Semi-supervised Learning:** Only a portion of the data provided to the ML models are labeled. The model attempts to optimally learn by combining the above two methods.
4. **Reinforcement Learning:** ML models apply a repetitive trial-and-error technique until their behavior improves towards one or more specified tasks, with or without manual tweaking of its developer(s) as the learning process progresses.

After the learning phase, which occurs during the training process, a model is deployed to make predictions on new data. The model's effectiveness in making accurate predictions is directly linked to the quality of its training. The process of providing inputs to the model to generate predictions is known as inference. In the context of classification tasks, which are the focus of this work, inference involves: (1) the compilation and deployment of a model to a target device, under a specific configuration to receive inputs and generate predictions; (2) the provision of a set of unknown/unseen inputs, not used in the model training; and (3) the extraction of model output, in the form of a ranked list of `K` predictions, ordered by decreasing certainty, across a set of possible outputs.

The work in this thesis is focused exclusively on testing related to the model deployment and inference process. Although investigating various aspects of the ML model training process is an interesting area of study, it falls outside the scope of this research.

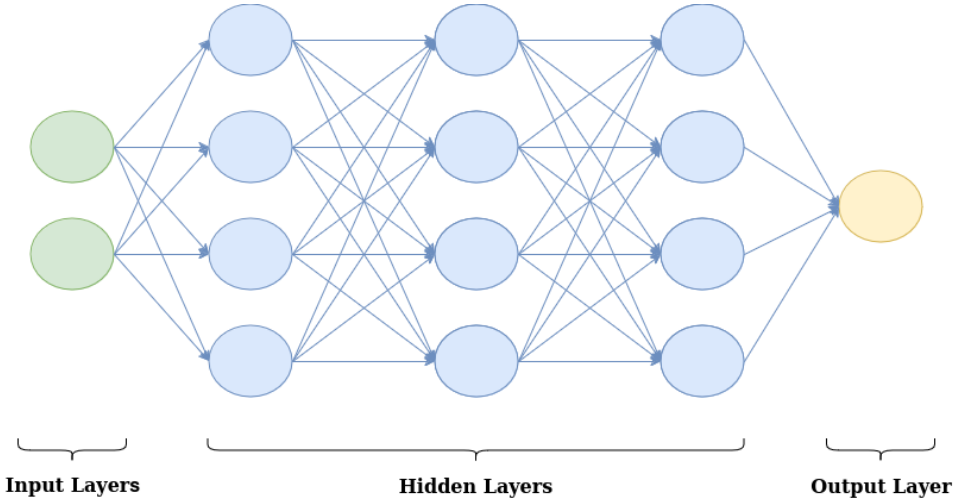


Figure 2.2: A Feed-forward Deep Neural Network.

2.2.1 Deep Neural Networks

One widely spread and utilized architecture of ML systems, are *Deep Neural Networks (DNNs)*. While there are multiple types of DNNs, we will focus on one specific category, the Feed-forward Neural Networks (FNNs). FNNs consist of a number of layers. The layers that receive the input are called *input layers*, the intermediate ones performing internal computations are called *hidden layers*, while the final output is retrieved through the *output layer*, which usually contains the result of an aggregation operation.

Each layer contains a number of nodes (neurons), and each neuron consist of (1) one or more inputs, (2) a number of numeric parameters that are utilized for computations in combination with the inputs in order to provide an output, (3) an output and (4) additional node properties, related with the configuration of the training process. The model parameter values are formed through the training process, while hyperparameters essentially contribute to the configuration of the model for that training process. In FNNs, the output of each neuron of a layer forms a weighted connection with all neurons of the next layer. A depiction of FNNs can be observed in Figure 2.2.

2.2.2 Image Recognition DNNs

DNNs are widely utilized in image recognition domains and assigned tasks such as classification, object detection, and semantic segmentation. A widely used architecture of DNNs, inspired by biological processes [84], is the Convolutional Neural Network (CNN). Its operation relies considerably on a set of computational layers called *convolutions*. Convolutions essentially perform a filtering operation over an input. They consist of the input (in the context of image recognition, usually a tensor representing an image or a portion of it), a feature detector (also called kernel or filter), and the purpose is to generate an output that provides information regarding where the feature is present inside the input - known as feature map, or activation map. In simple terms, the filter will examine the whole image in separate parts, in an attempt to detect specific features in them, performing a step-by-step process of moving over a fixed number of elements each time. This step value is called *stride*. The filter consists of a $N \times N$ two-dimensional matrix (quite commonly 3×3). On each step, the dot product between the sub-array under scan and the feature map is computed, and the output is stored in a separate array, and the process is repeated until the full set of features - called a feature map, is extracted as layer output. Figure 2.3 depicts this step-by-step process, obtained from Yamashita et al. [1], utilizing a 3×3 kernel and a stride of value 1 (as the filter "moves" 1 element at each time).

This technique enables a multilayer filtering approach for the model in order to classify inputs under specific categories (classification), or recognize multiple objects within an image or a scenery input (object detection), as well as the areas they occupy in this input (semantic segmentation). Subsequently, convolutions are widely used in models assigned with image recognition tasks.

2.3 Software Testing

Software testing is a process of *verification* (ensuring the system is correctly implemented) and *validation* (ensuring the system meets user needs) that a software system behaves in the intended way that its specification describes, under different scenarios and circumstances. Throughout the years, numerous software testing techniques have been developed, evolved and applied (such as unit testing and integration testing) [23], applied in both manual (e.g., developers running tests by hand) and

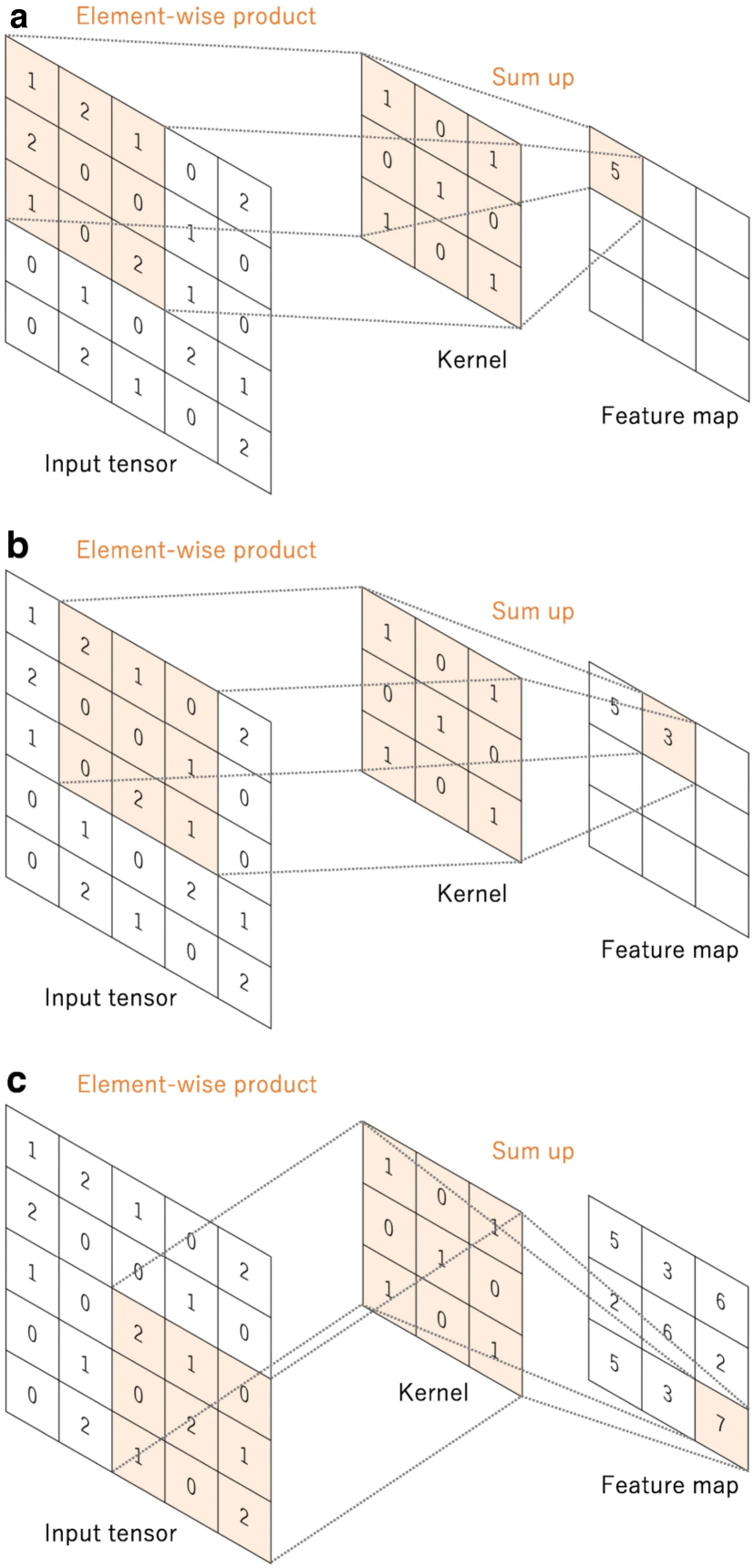


Figure 2.3: A step-by-step overview of convolution processing, as depicted in work by Yamashita et al. [1].

automatic (automated software systems with the responsibilities of running thousands of tests when specific criteria are met) manners. Those techniques are quite commonly utilized for software development and maintenance purposes, both in academic and industrial settings.

As software systems themselves, DNNs are also prone to errors introduced in their design and development process. In addition, DNNs are commonly applied in safety-critical tasks, such as autonomous driving, where a failure can be catastrophic [6]. Furthermore, verifying that DNNs not only work correctly, but within specific acceptable timing constraints, is crucial. However, DNNs cannot be considered "conventional". A Deep Neural Network (DNN) is a multi-layered model whose behavior is shaped by learning from large volumes of data, rather than by explicit programming. Unlike traditional software, where logic is explicitly defined, a DNN generalizes patterns from data through training, enabling it to make decisions based on experience instead of fixed rules.

DNN-based systems differ fundamentally from other types of software. In imperative or rule-based programming, logic is explicitly crafted by developers through sequences of commands and conditional flows. In contrast, a DNN derives its logic implicitly through exposure to training data. While data pipeline systems or Extract, Transform, Load (ETL) workflows process data in structured, predefined steps, DNNs adapt to the data and extract complex patterns, often in unstructured or high-dimensional forms such as images, audio, or text. Even compared to traditional statistical models or classical machine learning, which typically require manual feature engineering and produce interpretable outputs, DNNs stand apart by automatically learning hierarchical representations, though often at the expense of transparency.

In essence, a DNN represents a paradigm shift in software development—from designing rules to training behavior—prioritizing adaptability and data-driven generalization over explicit control and interpretability. For that matter, conventional testing techniques might not be effective. For instance, white-box testing (i.e., testing that is performed by having a full picture to the implementation and the inner mechanisms of a system) cannot be directly applied to DNNs, as their control flow is not sufficient to depict the knowledge obtained in their training phase, not allowing the definition of typical coverage criteria (e.g., statement coverage) [80].

In this thesis, we primarily utilize two conventional testing techniques, applied in the context of DNNs: *differential testing*(presented in detail in Section 2.3.3) and *mutation testing*(presented in detail in Section 2.3.5). Given that both techniques are mature in terms of conventional software, but also promising in the context of DNNs given the existing literature (e.g., DLFuzz [85] and DeepMutation [86]), our goal is to use them in order to further explore an aspect generally overlooked by the literature: *the computational environment of DNNs*.

System Performance

We refer to *system performance* as the measure of how effectively a system delivers its intended functionality, primarily evaluated by the correctness (or accuracy) of its outputs and the efficiency of its execution, typically expressed as execution time. *Correctness* indicates how closely the system's results match expected or desired outcomes, while *execution time* reflects how promptly the system processes inputs and produces outputs. In the context of DNNs, we also refer to execution time as *inference time*.

System Robustness

We refer to *system robustness* the ability of a system to maintain reliable and accurate performance when subjected to various challenges, such as unexpected inputs, environmental changes, or internal faults. This includes preserving the correctness of outputs and keeping execution times within acceptable limits despite such difficulties. In the context of DNNs, robustness means maintaining accurate predictions and reasonable inference times even when inputs are noisy, incomplete, or slightly altered (for example, properly recognizing objects in blurry images).

2.3.1 Fault Localization

Fault localization is the method of detecting the exact cause of a problematic behavior inside a system. Commonly referred to as "bug detection", it is an integral part of the software testing process. There are two main directions for fault localization: (1) manual and (2) automatic. Manual fault localization relies on the expertise of developers, testers, and QA engineers to inspect code and identify faults, often using tools like debuggers or memory inspectors. While effective due to deep code familiarity, it becomes less scalable as systems grow, highlighting the need for automation.

Automatic fault localization uses software to analyze and test code (statically or dynamically) to detect faults. It scales well, running thousands of tests quickly, but may produce false positives. Thus, combining it with manual techniques is often necessary for accuracy.

Various approaches have been proposed towards the direction of automatic fault localization in the context of Machine Learning. DeepFault [87] focuses on a spectrum analysis algorithm to identify problematic parts of the model that can be responsible for faults, and also proposes a method for adversarial input generation. DeepLocalize [88] converts the DNN model into an imperative representation and then performs dynamic analysis on its execution traces. On the other hand, NerdBug [89] utilizes traditional static analysis tools on DNNs in order to extract abstract information and then perform further analysis on them in order to identify bugs. Ghanbari et al. [90] propose a mutation-based fault localization technique, where it is measured to what extent mutants affect the passing/failing test cases in comparison to the original model, inspired by mutation-based fault localization applied in traditional software [91, 92]. In terms of DNN operators (e.g., Conv2D), Predoo [93] is a utility that performs input generation with the purpose of fuzzing DNN operators in order to unveil precision errors, while Chen et al. [94] propose a technique to discover bugs in DNN operators by using Metamorphic Testing. Finally, Yang et al. [95] focus solely on fuzzing the APIs provided by DL frameworks for Automatic Differentiation, a feature commonly utilized in ML systems for aspects such as backpropagation [96].

In this thesis, we contribute to this direction with `FetaFix`, a tool for automatic fault localization in DNN models affected by DL framework conversion. `FetaFix` localizes faults related to both model inputs and layers, offering a layer activation analysis module and various fault localization strategies. A detailed description of FetaFix is provided in Chapter 4.

2.3.2 Fault Repair

In the software development life cycle, once faults are detected, corrective actions are taken to resolve them based on the nature of the fault. Common actions include: (1) removing or correcting faulty code, such as eliminating duplicate database entries; (2) modifying the code to alter its control flow in order to prevent errors, like adding

safeguards to block file overwrites; and (3) adding fallback functionality, such as default settings when a configuration file is missing. Similar strategies apply to DNNs—for example, adjusting a problematic model parameter based on heuristics to improve performance.

These actions aim to fix faults and enhance software reliability. However, it's essential to verify that repairs do not introduce side effects (e.g., breaking previously working functionality). Thus, testing and validation are critical after any repair. Documenting changes and informing stakeholders also ensures coordination.

As with fault localization, fault repair can be manual or automated. Manual repairs benefit from developer expertise but may not scale well. On the other hand, automated tools designed to apply repairs add speed to the repair process, but they may sometimes be cumbersome and ineffective, potentially worsening the issues they attempt to repair or leading to catastrophic consequences, especially in safety-critical systems. Therefore, combining both approaches, followed by rigorous testing, is key to successful fault repair.

In terms of the research conducted towards the automatic fault repair of DNNs, Wu et al. [97] propose a concept for automatic DNN model repair, based on localizing and mutating defect layer weights affecting model predictions. Arachne [98] is a tool for automatic DNN repair by optimizing layer weights in order to improve the overall classifier model behavior, using differential evolution. As an expansion, Li et al. propose FedRep [99], a methodology that allows federated model repair based on Arachne, in a similar decentralized manner that federated learning methodology works. DeepCNN [100] proposes a fault localization and repair technique by utilizing a transformer encoder model that abstracts the token level of the DNN model code and attempts to localize faults and then repair them by applying code refactoring strategies. VeRe by Ma et al. [101] facilitates a DNN model repair process by utilizing formal verification by using linear approximation in order to calculate the repair significance of neurons across the model and optimize the parameters of those found to be problematic. Calsi et al. [102] also propose a methodology for search-based DNN fault repair dependent on the model state by applying intermediate searches across model neurons, with the purpose of maximizing a fitness function that considers potential values for suspicious model weights (calculated based on a set of misclassified inputs), in order to eventually repair problematic models. However, as a survey by Kim et al. [103] emphasizes, effective fault repair techniques in the context of DNNs are still immature and yet to be further developed.

Similarly to fault localization, we also contribute to automatic fault repair through `FetaFix`, detailed in Chapter 4. `FetaFix` implements six repair strategies targeting faults related to both model inputs and layers.

2.3.3 Differential Testing

Differential testing is a software testing technique utilized to compare the behavior of multiple different implementations or versions of a system, under the same inputs in order to test functionality present on both systems. The goal is to detect potential discrepancies in the outputs or the behavior of each version, which can potentially indicate the presence of faults or regressions. This technique can be useful when there is no formal specification that the system can be tested against for compliance, and there are different versions of the system available which can be compared against each other [104]. For example, consider a calculator application, with two releases one for "regular" use (including basic operations, such as addition and subtraction), and another for "scientific" usage, where additional functionalities are included to the regular version, for instance derivative calculation. Using differential testing, we could introduce a number of inputs related to the basic operators to both calculator systems and observe their behavior and output. If a discrepancy is found in their outputs, this could primarily indicate the presence of faults in one of the two versions of the system. However, differential testing has limited ability to detect faults that are present in both system versions, since it compares implementations against each other rather than against a formal specification.

Differential testing can also be used in situations where the same system operates in different configurations, but is expected to behave in the same manner. For instance, the behavioral consistency of an object detection DNN model utilized in autonomous vehicles can be examined by introducing inputs of different weather conditions (e.g., sun and rain) and comparing if the model properly identifies objects in the same manner in both conditions. In this context, systems such as DeepTest [105] and DeepRoad [16] generate adversarial inputs and examine the way object detection models behave. In fact, numerous works utilize this technique in the context of DNNs [106, 107, 108, 109], across different aspects. We further elaborate on this works on the next sections.

2.3.4 Fuzzing

Fuzzing [110] is the technique in which invalid, unexpected or random data are provided to a system in order to unveil vulnerabilities and bugs of various types, such as crashes, memory leaks and unexpected behavior. While our work does not primarily utilize fuzzing, numerous works that attempt testing within the computational environment utilize this technique, and therefore we define it here for the clarity of the reader.

2.3.5 Mutation Testing

Mutation Testing is a technique in which a software system is slightly modified with the goal of introducing small errors. If a test suite is sensitive enough to identify these small errors, it will probably also be capable of catching larger errors and faults. The slightly modified systems (called mutants) are semantically different from the original system, and are utilized with the purpose of examining the effectiveness of a testing suite. A good test suite should be able to "catch" all errors by failing all tests on the mutants that contain the introduced faults [111], "killing" them. Furthermore, a mutant that is not killed by the suite can reveal a weakness that the test suite has in detecting potential faults in this direction. For example, the addition of bias to a layer of a DNN model (which is an actual mutation examined in the literature [11]) will potentially result in the accuracy degradation of the model, and can be challenging for a test suite to catch (as such a change will subtly affect the model, without leading into a crash). While mutations are typically minimal (applied to specific parts of the software under test), they can also be combined and introduced in multiple areas of the system to more realistically simulate faults and evaluate the test suite under more challenging conditions.

Mutation testing is found to be effective, to some extent, in the context of DNNs [112]. DeepCrime [11, 12] introduces mutation testing in DNNs by simulating real DNN model faults and defining 35 mutation operators, relying on 3 empirical studies. DeepMutation [86] and DeepMutation++ [113] generate model mutants to assess the test data input quality for Convolutional and Recurrent Neural Networks. MuNN [114] explores the potential of mutation testing on DNNs related to the model characteristics, such as input and hidden neurons, with the purpose of checking both test adequacy and increasing model operation understanding. Cheng et al. [115] performed a study on code implementations of DL algorithms and detected faults by utilizing mutation testing. It is worth noting however that a study by Jahangirova

et al. [116] indicates that mutant killing is affected by the stochastic nature of the DNN training process, resulting in inconsistency across mutant killing for retrained instances of the same model. Finally, Tambon et al. [117] propose a probabilistic mutation testing technique in which the outcome of killing a mutant is treated as a Bernoulli trial. They further apply statistical distributions and computations to estimate the likelihood that a mutant will be killed.

2.4 Datasets & Architectures

DNNs cannot be tested using conventional software testing methodologies, such as functional and branch coverage, because they do not follow a traditional control-flow structure. Instead, the business logic of a DNN model relies primarily on a large set of tuned parameters, which are determined by the data used during the model's training process. The code itself mainly defines the model's structure and configuration. One aspect that has drawn considerable attention towards DNN model testing, is *adversarial testing:* researchers attempt to find effective was of generating inputs with the purpose of model crashes or accuracy degradation. This approach is easier to apply to DNNs, as it follows a black-box methodology and does not require deep knowledge of the complex decision-making process of a DNN. A continuation of this work involves DNN model fuzzing, often guided by coverage metrics. However, these metrics need improvement, as DNNs do not follow traditional control flow in their decision-making. Finally, limited work has been done on testing towards DNN model internals, as well as model architecture, primarily relying on mutation testing combined with input-based testing. We present all related work on the first two layers of the Deep Learning Systems Stack [79] in this section. Overall, existing research needs to be improved towards DNN model testing, in the directions of coverage criteria, adequate input selection and test oracle definitions for DNNs [28].

2.4.1 Surveys on DNN Model Testing

A number of surveys have been conducted in order to identify challenges and faults related to DNNs. In particular, Humbatova et al. [41] formed a taxonomy of faults for DNN models utilized in object detection. The authors surveyed commits, issues and pull requests from 564 GitHub projects and $9,935$ posts from Stack Overflow, and interviewed 20 researchers and practitioners and concluded faults related to the model, the GPU usage, the API, the tensors and inputs, and the training of DNNs. Huang

et al. [118] present a survey with focus on DNN safety and trustworthiness. Zhang et al. [43] comprehensively present existing testing techniques in machine learning by exploring a number of contributions in terms of correctness, robustness, and fairness, primarily focusing on model training and validation datasets. Bastani et al. [119] measure model robustness in accordance to constraints related to adversarial examples. Chen et al. [42] present a taxonomy of faults and difficulties of the DNN deployment process by sourcing a vast amount on StackOverflow posts. Islam et al. [120, 121] have conducted comprehensive studies on bug characteristics and the repair process of DNNs, by sourcing 970 repairs from Stack Overflow and GitHub for five popular DL libraries. They concluded in 15 bug fix categories related to model structure (e.g., adding layers, changing architecture, fixing dimensions), training configuration (e.g., tuning iterations), and data handling (e.g., data types, wrangling, and accuracy metrics). They also address compatibility issues like API contracts and library versioning. Our work, `FetaFix`, focuses on faults arising from DL framework conversion, as highlighted by Islam et al., specifically in the deployment and inference stages. We propose targeted strategies addressing both model structure and data handling.

It is very important to mention that these works emphasize that the computational environment aspects can raise considerable challenges towards DNN model deployment. Characteristically, Humbatova et al. [41] identified hardware faults as one of the basic taxonomy categories, as well as "API usage" as another crucial taxonomy; while the taxonomy includes "suboptimal model structure" as a category - an aspect correlated to model optimizations, as model graph optimizations attempt to address this issue. Also, Chen et al. [42] identify that "Model Conversions" are responsible for 26.5% of problems associated with DNN deployment on mobile devices (the highest percentage of the taxonomy concerning mobile devices, highlighting that the conversion process is error-prone) and 18.4% on browser platforms, while also 7.8% of issues in mobile are associated with DL libraries in general, and "Environment" (including library importing and version incompatibility) associated with 19.2% of the issues in browser platforms. "Inference speed" (an aspect associated with model optimization) was also a primary taxonomy in both categories, covering 3.9% of issues on mobile devices and 7.2% on browser platforms. These findings are in alignment with our findings (related to the experiments conducted using `DeltaNN`), which emphasizes the error-proneness of the model conversion process.

2.4.2 Data-Oriented Testing

Input Testing

Papernot et al. [122] propose a black-box adversarial attack technique in which the adversary trains a substitute model that approximates the target model using synthetic data generated via query responses. Adversarial examples crafted on the substitute model are then transferred to the target model to identify misclassifications. Carlini et al. [123] propose a methodology of introducing adversarial sections in image inputs to misclassify defensively distilled models. Wei et al. [44] suggest an approach for introducing adversarial stickers on physical objects to achieve model misclassification, while DeepBillboard [124] similarly attempts to exploit real-world billboards by introducing adversarial patterns. Brendel et al. [125] emphasize the generation of adversarial inputs with minimum boundary changes in comparison to the original input. Wicker et al. [126] propose a black-box methodology for effective adversarial input generation that follows a turn-based stochastic game approach. DeepEvolution [127] performs a search-based testing approach, relying on metaheuristics generate effective test inputs. RobOT [128] proposes a metric associated with adversarial input generation involved in model retraining to increase DNN model robustness, while DeepFool [129] generates small input alterations ("perturbations") on inputs of classification models in order to achieve misclassification. DeepTest [105] modifies images using linear & affine transformations, and generates inputs simulating different weather conditions and phenomena to stress-test DNNs utilized for autonomous driving. DeepRoad [16] applies in the same context, while using GAN-based metamorphic testing that simulate extreme weather conditions, such as heavy rain and snow. Going beyond the image recognition domain, DeepStellar [130] focuses on the generation of adversarial inputs for Recurrent Neural Networks (RNNs) by analyzing the state transition of the RNN model, while DeepCruiser [131] follows a coverage-guided test generation approach in order to unveil faults in RNNs. For a more comprehensive overview of adversarial inputs of DNNs, we refer the readers to a survey [132].

Coverage-Guided Testing

As mentioned earlier, in conventional software, input-oriented testing techniques are commonly combined with coverage metrics to enhance effectiveness and guide the testing process. However, conventional software coverage metrics (e.g., function and branch coverage) are not suitable for DNNs [26]. Since the business logic of a DNN model is primarily data-driven, shaped by the data used during training and relying on a large number of parameters rather than explicit code, new coverage metrics must be defined. As a result, there have been contributions that propose coverage metrics specifically designed for the context of DNNs. Li et al. propose DeepGauge [133], a set of five coverage criteria for DNNs - both in neuron and layer levels. DeepXplore [45] applies white-box testing by measuring neuron coverage, identifying similar DNNs for cross-reference and generating adversarial inputs to detect faults, while FuzzGAN [134] utilizes a Generative Adversarial Network technique combined with neuron coverage to generate adversarial inputs against DNNs for the detection of flaws. TensorFuzz [135] utilizes fuzzing related to DNN model layer activation coverage by feeding inputs in TensorFlow model graphs with the aim of bug detection. Test4Deep [136, 137] also attempts to perform a white-box testing approach that generates test data that cause model faults. This is performed by measuring and improving neuron coverage, by triggering inactive neurons in a DNN model under test and observing how the model behaves upon these activation triggers. Sun et al. [80] propose a methodology for coverage criteria and test generation, based on Modified Condition/Decision Coverage (MC/DC) criterion, used in conventional software. DeepPath [138] is also a utility for path-driven coverage measurement and test generation via automatic path exploration. DLFuzz [85] attempts to minutely mutate inputs to improve neuron coverage. Adapt [139] also attempts white-box testing by adaptively determining a set of neurons potentially responsible to faults in order to compute gradients to generate adversarial inputs and maximize coverage, as the testing process evolves. DeepPath [138] is also a utility for path-driven coverage measurement and test generation via automatic path exploration. DeepGini [140] compares the class probabilities across input predictions in order to quickly detect misclassified test cases, based on the intuition that a test input has a high chance to be misclassified if the prediction probabilities are similar for each class. DeepH-

unter [141] utilizes metamorphic mutation strategies by generating test cases following a diversity-based seed selection process while leveraging multiple coverage criteria. Finally, DeepConcolic [142] utilizes coverage-based test case generation combined with concolic execution in order to test model correctness.

2.4.3 DNN Model Architecture Testing

Most of the works related to DNN model architecture testing involved (1) white-box testing, by analyzing the model state (e.g., activations), (2) mutation testing of the model to detect defects, and (3) hybrid testing - combining multiple testing techniques. While we briefly present existing works for (1) and (3), mutation testing is presented in detail in Section 2.3.5, in association with our contribution related to mutation testing of AI compilers with relation to hardware acceleration target code.

Sun et al. explore the potential of coverage-based concolic testing on DNNs, by proposing (1) a technique that uses existing coverage criteria combined with concolic testing to maximize test coverage [143], and (2) a suite named DeepConcolic [142] that utilizes coverage-based test case generation combined with concolic execution in order to test model correctness. Ma et al. [144] propose DeepCT, a combinatorial testing suite for coverage-guided test generation, relying on neuron state and activations. Deokuliar et al. [145] attempted to combine test cases generated by DeepXplore with DeepMutation [86], a mutation testing utility (presented in detail below), to further discover improved test inputs based on efficacy metrics. Wang et al. [146] propose a methodology that examines model robustness against adversarial inputs combined with model mutations, examining how DNN models behave in the presence of faults. Finally, Lu et al. propose RGChaser [147], a tool that utilizes mutation testing on model inputs with Reinforcement Learning in order to detect inputs that lead to model misclassifications.

2.4.4 DNN Model Inspection

A number of contributions propose methodologies for model inspection that can be utilized for bug detection. Grad-CAM [148] enables visual explanations for CNNs by examining the gradients flowing into the final convolutional layer to form heatmaps, highlighting the input sections that contributed in the model to the "target concept" (e.g., predicted classification label). DVTest [13] is a visual testing utility based on metamorphic testing and DNN feature map visualization, while also enabling neuron freezing and retraining. DeepInspect [149] automatically detects confusion and biases

related to the classes of the models utilized for classification tasks. DeepBase [150] is a tool developed for the inspection and analysis of DNNs. It allows users to define high-level behavioral hypothesis functions, which represent expected patterns or behaviors within the model. The tool then enables the observation and analysis of statistical relationships in the model's internal processes, identifying whether the model's behavior aligns with the hypothesized functions. This helps in understanding and verifying how the model makes decisions. Towards DNN analysis, Ma et al. [151] also utilize DNN debugging by applying differential state analysis across layer heatmaps for different inputs, with the purpose of bug detection. Sun et al [152] propose a statistical fault localization methodology, by utilizing the heatmap explanations from DNN inputs of the model. Gopinath et al. [153] propose DeepCheck, a methodology that enables lightweight symbolic execution and analysis of DNNs for validation purposes. In terms of program orchestration, SynEva [154] utilizes program synthesis in order to generate an oracle-alike mirror program of a model and test it in different scenarios to evaluate its effectiveness, essentially enabling differential testing with the purpose of model behavior observation.

2.5 The Computational Environment

Figure 2.1 gives an overview of the typical layers in the Deep Learning Systems Stack [79]. As aforementioned, much of the existing work on Deep Learning model robustness has focused on testing robustness with respect to the top two layers, *Datasets* and *Models*. However, exploring robustness with respect to the computational environment (last three layers of Figure 2.1) has received little attention. We present all related contributions in the following sections.

2.5.1 Deep Learning Frameworks

Deep Learning (DL) Frameworks (the third layer in Figure 2.1) provide utilities such as model definition, training, and inference to ML engineers. Quite commonly, they consist the basis on which DNNs are built, as they provide ready-to use implementation for crucial DNN model building block functionality (e.g., implementation of convolutions) and features (e.g., optimizations and hardware acceleration functionality). The

most common of these DL frameworks are strongly supported by active communities of developers and large corporations (e.g., Google and Meta). However, due to the complexity of the tasks these frameworks are used for, thorough testing is essential. This thesis aims to contribute to this direction.

Related Work

Guo et al. [155] conducted an empirical study highlighting that differences across DL framework implementations can lead to considerable issues, such as degradation in model accuracy. In addition, some research has been conducted in the direction of analysis of model training and inference performance [156, 157, 158, 159, 160].

Existing Studies: A recent survey [161] explores various parameters and their effect towards model accuracy and execution time. Zhang et al. has also conducted a study with relation to TensorFlow bugs [162], determining challenges and suggesting potential fault mitigation and bug detection strategies, such as model accuracy comparison with fixed thresholds between iterations, replacing hyperparameters, examining distribution of variable values and switching the training dataset. We draw inspiration from this work for `FetaFix`, as their findings and proposed mitigation strategies align with our methodology. Nejadgholi et al. [163] also conducted a study related to oracle approximation practices utilized by developers when testing DL frameworks and libraries.

Automated Fault Detection in DL Frameworks: With respect to the automated fault detection of DL frameworks, there are some works aiming to detect and localize inconsistencies between models sourced from different DL frameworks. A survey by Kim et al. [164] revealed that building DNNs with different DL frameworks can lead to different accuracy of the top-1 prediction, by performing inference on models built on Keras, PyTorch, and MXNet using Dogs vs. Cats dataset from Kaggle and detecting different F1 value and accuracy. Another contribution by Florencio et al. [165], indicated differences in terms of execution times, with PyTorch achieving better performance, even though TensorFlow achieved a greater GPU utilization rate. Chen et al. [166] performed a large-scale study of common bugs across common DL frameworks, obtaining 12 major findings and proposing potential action guidelines for their mitigation. CRADLE [17] performs an execution analysis on the model graph in order to detect faults, while LEMON [18] utilizes the metrics used by CRADLE and applies mutation testing in order to identify issues in the model. FAME [167] is based on LEMON but enables API mutation and optimizations for layer and weight mutation, guided by the detection of inconcistency faults. Similarly,

Audee [168] aims to detect logical, not-a-number bugs, and crashes by applying an exploratory approach in combination with mutation testing. MMOS [169] uses multi-step mutation testing to identify faults in DNNs, helping to improve their robustness and reliability. COMET [170] applies a coverage-guided model generation for mutation testing purposes focusing on layer API calls of DL frameworks. Luo et al. [171] propose a technique that applies Monte Carlo Tree Search (MCTS) combined with operator-level coverage in order to effectively fuzz DNN models, by implementing six model mutations concerning model parameters, structures and data. DeepChecks [172] also tests and checks data integrity and distribution utilized by the model training process as well as predictions on model performance. UMLAUT [173] utilizes debug heuristics defined by experts related to model behavior and structure in order to perform fault localization. DeepDiagnosis [174] also attempts fault localization by examining the DNN training and model execution process against eight DNN-oriented fault heuristics. DeepFD [175] also follows a search-based fault diagnosis approach by training a DNN under test while extracting a number of fault diagnostic features and then comparing them with those of pretrained diagnosis models, with the purpose of fault localization. In terms of numeric precision errors, GRIST [176] attempts to utilize gradient back-propagation combined with static analysis in order to detect numerical faults (such as `NaN`, `Inf`, and invalid values). Finally, ACETest [177] utilizes a technique for automatic constraint extraction for model inputs and uses it to generate test cases automatically, focusing on DNN operators implemented in DL libraries.

DL Framework Fuzzing: DeepCov [178] utilizes coverage-guided fuzzing with focus on DL frameworks in order to detect faults, by considering the layer-edge coverage and utilizing LEMON's Layer Addition (LA) operator to generate model mutants that achieve more DL framework API calls. Gandalf [179] utilizes a generic context-free grammar with reinforcement learning in order to generate diverse test cases for DL frameworks, while also defines a set of metamorphic relations to generalize tests across DL libraries. TitanFuzz [180] utilizes a generative LLM in order to generate seed programs with the purpose of fuzzing DL libraries. On the other hand, FreeFuzz [181] performs DL framework fuzzing by mining code snippets and models from open source repositories. DeepREL [182] tests DL libraries by considering that for a DL library under test, there might be relational APIs that share input/expected output pairs. It identifies such pairs coming from a set of already invoked APIs and uses them to fuzz the relational APIs under test. DocTer [183] utilizes DL framework documentation in order to perform fuzzing on DL libraries with the purpose of bug

detection. EAGLE [106] utilizes differential testing across DL frameworks for models of the same architecture, specifically PyTorch and Tensorflow, focusing on issues related to different API implementations across DL frameworks. As an extension, CEDAR [184] attempts to combine DocTer and EAGLE methodologies with the intention of finding new faults. Muffin [107] also combines fuzzing with differential testing by enabling the construction of DL models with the purpose of fuzzing different DL libraries, while also comparing the behavior of each one of them across the others.
Other DL Framework Testing Utilities: In terms of effective mutant selection, Feng et al. [185] propose a methodology for mutant operator reduction by calculating decision boundary distances across the original DNN model and each of the mutants. Finally, regarding test maintenance, DiffWatch [108] focuses on detecting automatically changes in DL libraries that affect predefined test cases used for differential testing across DL libraries.

Our Contributions

In this thesis, we primarily consider four widely used DL frameworks: *Keras*, *PyTorch*, *TensorFlow (TF)*, and *TensorFlow Lite* (*TFLite*). We selected these DL frameworks for our experiments due to their widespread adoption in both industry and research, making them representative of commonly used DL frameworks. As of the time of writing this thesis, the paper on TensorFlow and TF Lite has been cited $47,461$ times and their GitHub repository has been starred 190K times, the paper on PyTorch has $56,798$ citations and its GitHub has been starred 89.8K times, and the paper on Keras has $22,807$ citations and its GitHub repository has been starred 63K times.

Keras [47] is a high-level DL framework that provides APIs for effective DL usage. It acts as a high-level interface for TensorFlow.

PyTorch [46] is an open source ML framework based on the Torch [186] library. It supports hardware acceleration for tensor computations.

TensorFlow (TF) [51] is an open-source DL framework developed by Google, widely used for training and inference of DNNs.

TensorFlow Lite (TFLite) [51] is a lightweight version of TF, and part of the full TF library, focused only on the inference of DNNs on mobile and lightweight devices.

We conducted our experiments using the ILSVRC dataset [69] (a dataset used extensively for benchmarking of well-known classification and object detection models, such as ResNet [66]), and comparing the top-1 class prediction output for each input across all DL frameworks for each model (which indicates the class with the highest prediction probability for an input in a classification model, and thus receives the greatest emphasis).

Our experiments were proven to agree with the results of the aforementioned studies, presenting differences both in predictions and execution times. However, our work goes beyond comparing DL frameworks by exploring in-depth the effects of the conversion process across DL frameworks (e.g., converting a pre-trained model from PyTorch to Keras) - a process commonly utilized by researchers and developers for portability and extensibility purposes. In addition, existing works employ methodologies related to ours, such as DeepFD [175], which uses a search-based approach for fault diagnosis and localization in deep neural networks, and EAGLE [106], which adopts a differential testing strategy for deep learning frameworks. However, none of these approaches explore the challenges associated with DL framework conversions or propose any solutions towards their automatic fault localization and repair.

2.5.2 DL Framework Conversions

DNN models are converted from one DL framework to another, for two primary reasons: (1) portability - so that a model can be deployed on devices of different computational capabilities; and (2) supported features and/or operations, such as extended optimization support or better profiling options. The conversion process enables transferring a model between deep learning frameworks without having to rebuild its graph architecture and retrain it from scratch, both of which are typically costly in terms of time and computational resources. The conversion process is conducted by providing a pre-trained model built on one DL framework (e.g., PyTorch). We call this model *Source*. Then, the conversion tool will use *Source* in order to generate the same model, but built in a new DL framework (e.g., Keras). We call this model *Target*. There is a wide variety of open-source tools for this purpose [53, 54, 56, 57], and the conversion procedure is performed automatically. However, the conversion process might involve multiple converters and might also require the model to be built using an intermediate representation (e.g., ONNX [58]). For example, the conversion of a model from PyTorch to Keras might require a two-step conversion: (1) PyTorch to ONNX, using the native converter of PyTorch to generate ONNX; and (2), the

`onnx2keras` [56] converter, in order to convert the ONNX intermediate representation to Keras. In addition, in cases where a converter tool might be found inadequate or an automatic conversion between DL frameworks might be unsupported, developers might choose to manually convert a model from one DL framework to another (for example, as we can see in [187]), a process that can be prone to errors.

Related Work

None of the aforementioned approaches focus on faults encountered during DNN model conversions. In this context, a variety of tools exist for the purpose of DNN framework conversions, such as `tf2onnx`[54], `onnx2keras`[56], `onnx2torch`[57] and tflite2onnx [55], as well as native APIs of TFLite and PyTorch. However, the error proneness of the process is overlooked in the literature, with limited research being conducted to this direction. Openja et al. [188] published a study that emphasizes the challenges of the conversion process, while Jajal et al. [189] also conduct a survey on ONNX model converters, focusing on `tf2onnx` and `torch.onnx` [46] tools, and concluding that node conversion in one of the major aspects towards problematic conversions and that semantically incorrect models also play a non-negligible role to them as well. Liu et al. [53] propose MMdnn, a framework that automates the conversion process between DL frameworks and addresses challenges like missing operators, inconsistent tensor layouts, unsupported padding, and incompatible argument types. However, it does not account for configuration-related issues, such as input preprocessing, or other important factors like layer parameters and computation graphs, which can also lead to output incompatibilities during conversion. This thesis presents a suite that addresses these additional aspects. It is also worth noting that MMdnn is currently in a deprecated state, with its last update on August 2020. In addition, our efforts with testing MMdnn were unsuccessful as it was built with older, deprecated versions of DL frameworks (e.g., PyTorch v0.4.0 while the current version is v2.5.1). In an attempt to compare MMdnn with our work, we identified a potential overlap of only 15 potential faults out of the 462 we repaired with `FetaFix`, based on the results reported in [53]. Finally, MMdnn supports conversions over a restricted set of DL frameworks (primarily CNTK, CoreML, Keras, MXNet, PyTorch, TensorFlow, Caffe, and DarkNet) that is not currently maintained or updated while our approach is generalizable to model conversions between any pair of DL frameworks.

Our Contributions

As aforementioned, Chen et al. [42] identified that model conversions can be responsible for a considerable amount of problems related to DNN model deployment (26.5% of problems associated with DNN deployment on mobile devices and 18.4% on browser platforms). We examined the correctness of the conversion process using `DeltaNN`, and conducting experiments across all permutations of 4 popular DL frameworks, for 3 widely-used classification models. Our results highlighted the error-proneness of the conversion process, to a much greater extent than Chen et al. indicated, finding 26 out of the 36 attempted conversions (72.2%) to be problematic to at least some extent, leading into crashes or accuracy deviations between the source and target model variants.

In terms of fault localization and repair of DL framework conversions, limited work has been conducted in the literature. Until recently, the most valuable contribution to this direction, is MMdnn [53], which, however, comes with a number of caveats and limitations. Our work [76, 77] is the first to effectively localize and repair DNN models converted by a variety of DL framework converters, using strategies based on existing faults and issues. As mentioned earlier, MMdnn is the main work related to ours, that provides conversion repair strategies. However, our approach offers a more comprehensive set of repairs, including considerations for layer parameters, the computational graph, and input preprocessing. Additionally, MMdnn is deprecated and supports only a limited set of DL frameworks, while our framework-agnostic approach does not rely on specific converters, enabling the repair of models converted using proprietary converters or manual conversion. Our work can also be adapted to any new converters that use the ONNX format as an intermediate representation. Notably, our work has minimal overlap with the faults identified by MMdnn, an aspect that demonstrates its potential and effectiveness.

2.5.3 Compiler Optimizations

In order for DNNs to perform effectively across the demanding tasks they are assigned to accomplish, they need to be optimized so that they can perform faster while preserving their efficiency. This aspect (part of the fourth layer in Figure 2.1) is crucial in resource-intensive, real-time systems (especially those handling safety-critical tasks, like DNNs used in autonomous vehicles), where they must operate under specific temporal constraints.

Related Work

A recent study [190] examined bugs introduced by different DL compilers. Incorrect optimization code logic accounted for 9% of the bugs introduced by compilers. Other compiler bugs presented in the study include misconfiguration, type problem, API misuse, incorrect exception handling, and incompatibility. DiffChaser [109] explores the application of differential testing across optimized and unoptimized DNN model variants, but does not generalize to multiple computational environment aspects. It considers only output label correctness and focuses on the generation of test inputs that generate different results across model variants rather than repairing the model or explaining potential causes of difference. Xiao et al. [19] propose a metamorphic testing methodology for AI compiler bug detection, including issues related to optimizations.

Our Contributions

In this thesis, we investigate the robustness of DNN models under compiler optimizations, with a primary focus on graph-level transformations. Using `DeltaNN` [70], we conduct a set of experiments that consider graph-level optimizations, in combination with deployment across different GPUs. We further elaborate on these experiments in Chapter 3. Our motivation comes from the fact that such optimizations are applied as integral parts of many widely-used AI compilers, optimizers, and frameworks, including Apache TVM [60], MLIR [61], and the ONNX Optimizer [191]. Representative examples of these transformations include, but are not limited to, operator fusion, elimination of common sub-expressions, modifications to data layouts and memory access patterns optimized for target hardware, and precision-reducing techniques such as fast-math.

2.5.4 Hardware Acceleration Devices

DNNs typically consist of layers, responsible for computations across multi-dimensional matrices - called tensors. Given that the size of these tensors can range from a few thousands to millions of elements, it is vital that for the effective and practical usage of the model, such computations are parallelized. For that purpose, hardware acceleration devices such as GPUs and TPUs exist, specializing in computations across vectors and tensors. With the recent rise of ML, industrial and research emphasis has been given towards the implementation of more powerful hardware acceleration

devices, with companies such as NVIDIA [192] and AMD [193] implementing cutting-edge devices specializing in AI. However, the capabilities, computational capacity and architecture can vary greatly across devices, which can potentially affect the model correctness and performance. In this thesis, we consider this aspect by conducting experiments on 4 different hardware acceleration devices, of varying capabilities - from GPUs integrated to servers down to mobile devices. Our aim, is to explore and unveil potential accuracy, or execution time inconsistencies across devices. In particular, we utilize them in two directions: (1) by conducting differential testing and comparing accuracy and execution times, using `DeltaNN` (Chapter 3), and (2) by performing mutation testing and robustness check across them, using `MutateNN` (Chapter 5).

Related Work

Hardware Acceleration Faults: For the hardware layer in Figure 2.1, a study forming a taxonomy of faults encountered in DNNs used in object detection [41] revealed *GPU related bugs* to be one of the five main categories of faults in DL tasks like object detection. The study, however, did not explore the impact of these bugs on model performance. To this direction, Omland et al. [194] attempted API-based fault simulation of hardware faults related to mathematical computations in DNNs. Also, Rahman et al. [195] explore the robustness of DNNs on transient hardware faults by proposing TensorFI, a fault injector utility for that purpose. Finally, Chaudhuri et al. [196] propose a method that uses a two-tier DNN in order to detect structural faults in processing elements of systolic arrays in hardware accelerators. In addition, there are numerous works related to compiler fuzzing and mutation testing. In order to detect hardware acceleration bugs that could lead to performance degradation, Moussa et al. [22, 197] propose a methodology related to identifying weight faults at the bit level associated to hardware performing DNN calculations. In contrast to our work, their contribution is not related to errors introduced from miscompilations (compiled code that behaves differently from the intended semantics of the source program due to one or more compiler faults). GenCoG [198] focuses on generating valid and expressive TVM computation graphs by defining a domain specific language for constraint definition and applying concolic constraint solving. Li et al. [21] propose a methodology to uncover hardware-related faults by generating fault-sensitive, adversarial tests.

AI Compiler Testing: HirGen [20] is a tool for fuzzing DL compilers that focuses on the high-level model intermediate representation (IR) of TVM. It detected 3 inconsistency issues related to LLVM and CUDA implementations across devices. However, none of these inconsistency issues were related to issues with conditional expressions in kernel code, which our work detected and highlighted.

A common approach of AI compiler testing utilized in the literature, is by applying fuzzing. TVMFuzz [199] is a tool that allows generic-purpose TIR fuzzing on Apache TVM. It detected primarily crash and wrong bit manipulation bugs, failing to detect non-crashing effects on model correctness related to miscompilations across devices, which our methodology focuses on. Tzer [81] is a coverage-guided fuzzer that applies modifications on models compiled in Apache TVM utilizing Tensor IR (TIR). NNSmith [200] focuses on the generation of valid DNN graphs that can be used to examine the compiler for bugs in a similar manner with grammar-based fuzzing techniques for traditional software programs (e.g., [65, 201]), and MLIRSmith [202] integrates NNSmith's fuzzing capabilities in MLIR [61] in a random manner, while Suo et al. [203] follow an operation dependency approach to guide the fuzzing process in MLIR. SYNTHFUZZ [204] combines grammar-based fuzzing with custom mutation synthesis in order to effectively fuzz MLIR dialects. DeepDiffer [205] applies low-level IR fuzzing in a coverage-guided manner and comparing results across different DL compilers. Finally, Su et al. [206] fuzz DL compilers in the context of dynamic features (e.g., dynamic control flows and closures), detecting bugs in PyTorch and its underlying tensor compiler. However, all these approaches primarily focus on compiler crashes and do not address accuracy inconsistencies that may arise across different hardware acceleration devices. Our work aims to contribute to this gap by exploring such inconsistencies.

Our Contributions

We examine the effects of the hardware layer to DNN model correctness and execution time performance in two directions: (1) by using `DeltaNN` [70] as one of the computational environment aspects under test, conducting a set of experiments (Chapter 3) and (2) by combining mutation and differential testing in the direction of hardware acceleration devices, using `MutateNN` [78] - the first work, to the best of our knowledge, that attempts to explore the effects of faults in the target code across different hardware acceleration devices - a result of bugs introduced by developers, but also coming from bugs in the AI compiler. Considering that real faults [41]

can prove quite challenging to detect when running DNNs in a real-life production scenario [82], we conduct experiments across four hardware acceleration devices of varying capabilities to assess model behavior and robustness in environments where they are typically deployed.

Our proposed methodology is related to behavioral faults detection, primarily focusing on the context of faults present in target code, resulting into behavioral discrepancies across hardware accelerators. Our work is inspired by (1) a well-established taxonomy of faults [41], where GPU issues are identified as a primary cause of DNN model faults, (2) a survey [207] highlighting that GPU computations are among the primary causes of challenges that DNN developers experience, and (3) the evolution of AI compilers (e.g., Apache TVM [60]) and compiler frameworks (e.g., MLIR [61]) that automatically generate target code but also the release of frameworks that provide APIs for target code development (e.g., Triton [73]).

2.6 Machine Learning Tools

2.6.1 Open Neural Network Exchange (ONNX)

ONNX [58] is an open standard for machine learning interoperability that allows the representation of DNNs using a common set of operators and file formats to enable interoperability with a variety of tools and systems. It provides an API to analyze the DNN model graph, hyperparameters, and parameters. The model is represented by a graph of operations (or nodes), each containing a set of properties. These operations are executed sequentially, propagating data from one layer to its subsequent ones. Throughout the years, ONNX has evolved producing newer versions of the standard, modified from the past. Each standard version is attributed an `opset` number. Due to the popularity of ONNX and the versatility of its API, in this thesis we perform automatic fault localization and repair at this representation level - retrieving ONNX versions of the *Source* and *Target* model, analyzing them and repairing them. For that purpose, when we refer to *Source* and *Target* we implicitly refer to the ONNX variants of the DNN models under test.

Table 2.1: Inference accuracy of native models on the ImageNet dataset.

DNN Model / Framework	PyTorch	Keras	TF	TFLite
ResNet101	81.9	76.4	77.0	77.0
InceptionV3	77.3	77.9	78.0	78.0
MobileNetV2	72.2	71.3	71.9	71.9

2.6.2 Models and Inputs Selection

In this thesis, we primarily utilize three widely used CNN models: MobileNetV2 [71], ResNet101V2 [66], and InceptionV3 [72]. These models are widely known and extensively used for classification and semantic segmentation operations, and are the "backbone network" for other tasks such as object detection [208]. All three models have native definitions within the DL frameworks under study. The accuracy of the native version of each model is shown in Table 2.1. We selected CNNs for our work, as they are widely used for image recognition tasks, while their size makes them tractable in terms of experiments. However, it is worth noting that transformer-based architectures [209] have also begun to provide competitive results in recent years [210, 211] in the context of computer vision, however, their usage is still on the rise for real-life applications and scenarios.

It is expected that the same DNN model may have varying accuracy between DL frameworks, as each framework will define and train their own version of the model from scratch, thus producing different parameters (since training is stochastic); and there may even be small differences in the graph definition (e.g., different padding parameters). We observe that TF and TFLite models have the same accuracy, suggesting that the latter models were converted from the former by developers.

In terms of input selection for our work, a common benchmark for Perception AI models is the ImageNet image classification dataset [69], which requires assigning one of 1000 possible class labels to RGB images of 224×224 pixels.

2.6.3 Apache TVM

Apache TVM [60] is an end-to-end machine learning compiler framework for CPUs, GPUs, and other accelerators. It generates optimized code for specific DNN models and hardware backends, allowing us to import DNN models from a range of DL frameworks. It also provides profiling utilities such as per-layer inference times. A simplified representation of Apache TVM can be seen in Figure 2.4. TVM's support of several DL frameworks, optimization settings, and hardware accelerators made it a suitable choice to explore different environment parameters in our experiments. TVM provides direct importers for models from most popular DL frameworks, which load said models as a TVM computation graph. The first level of optimization available in TVM is graph-level optimizations, which is the focus of this thesis. These optimizations impact the full DNN model and include operator fusion (e.g., batch normalization, activation functions), elimination of common sub-expressions, and potentially unsafe floating-point precision optimizations such as fast-math. TVM also supports optimizations for a given operation type (e.g., convolutional layers, matrix-multiplications) such as loop tiling, loop re-ordering, loop unrolling, vectorization, auto-tuning [212], and auto-scheduling [213], among others. It also supports third-party libraries such as cuDNN [214] and the ARM Compute Library [215].

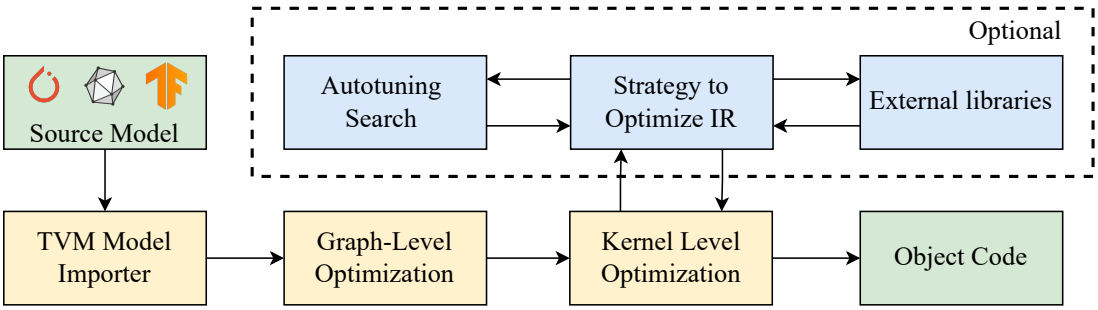


Figure 2.4: Overview of DNN compilation in Apache TVM.

TVM features a very powerful debugger, which allows to export metadata regarding the DNN model inference (such as layer activations, execution times per-model layer), as well as metadata about model structure and parameters (each exported in separate metadata files) while also providing a high-level API for their analysis.

2.7 Summary

In this section, we provided details about all the necessary term definitions and concepts that we utilize in this thesis. The reader is introduced in Sections 2.2 and 2.3 on the most important concepts concerning machine learning and software testing, along with all existing related work. Section 2.4 further elaborates on testing in the scope of DNNs, while Sections 2.5 and 2.6 present information and existing research with regard to the main computational environment aspects considered and the main AI tools used in this thesis.

Chapter 3

Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models

3.1 Introduction

Much of the existing literature for assessing robustness and safety of image recognition models has focused on testing the Deep Neural Network (DNN) structure and addressing biases in the training dataset through adversarial examples and data augmentation [16, 85, 105]. However, the impact of computational environment aspects related to the DNN model deployment process, post training, has not yet been explored. In particular, existing techniques fail to consider model output errors and unexpected execution time performance degradations, that could potentially be caused by interactions of the DNN model with the underlying computational environment aspects – such as processes related to Deep Learning (DL) frameworks that the models are built on (e.g., conversion of a model built in PyTorch to TensorFlow Lite in order to be deployed to an edge device), compiler optimizations applied to the models to improve their inference time (e.g., operator fusion, loop unrolling, etc.), and the hardware platforms they run on (e.g., CPUs, GPUs, etc.). Figure 3.1 shows potential sources of error in the computational environment when a DNN model is deployed. These environment aspects are important considerations in DNN model maintenance and evolution.

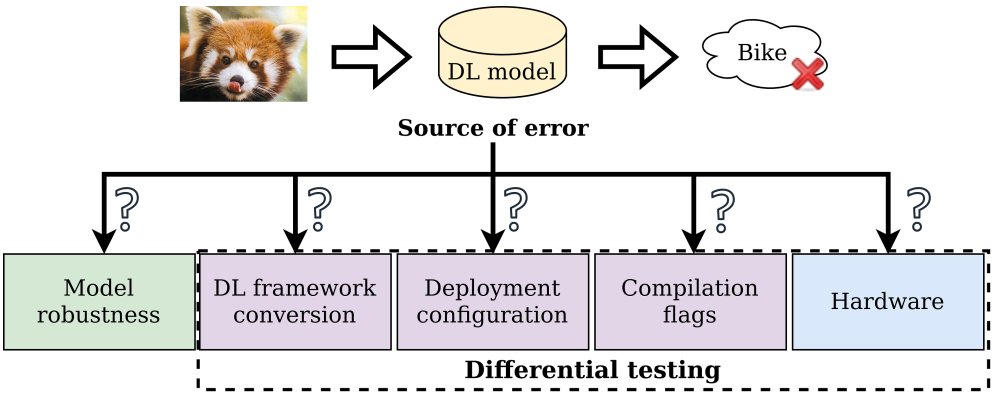


Figure 3.1: Possible sources of errors when deploying DNN models.

To understand the impact of the computational environment on model deployment, we present a differential testing framework, `DeltaNN`, that helps evaluate the robustness of image recognition models to changes in specific aspects of the computational environment (Figure 3.2). `DeltaNN` takes as input a trained DNN model defined in a given DL framework and produces different implementations by changing the following parameters in the computational environment:

DL frameworks: transforming a model defined in one DL framework to the model format of another framework. Studying the impact of model conversion between DL frameworks is important, as developers may often convert their models to different DL frameworks (e.g., TensorFlow Lite) to support resource constrained environments on mobile and IoT devices [216, 217]. Automated conversion processes suffer from faults, mainly caused by unsupported operations in the target framework, or by the converter. We generate different implementations of every trained model with several popular DL frameworks.

Compiler optimizations: considering different levels of compiler optimizations with each generating a distinct code implementation. The focus of our experiments is on graph-level optimizations such as operator fusion, eliminating common subexpressions, data layout transformations for better memory utilization and access patterns on target devices, or potentially unsafe optimizations such as "fast-math". Compiler optimizations are expected to improve model inference time performance, sometimes at the cost of model accuracy. We study this parameter to understand the extent of the impact on model correctness and inference time performance.

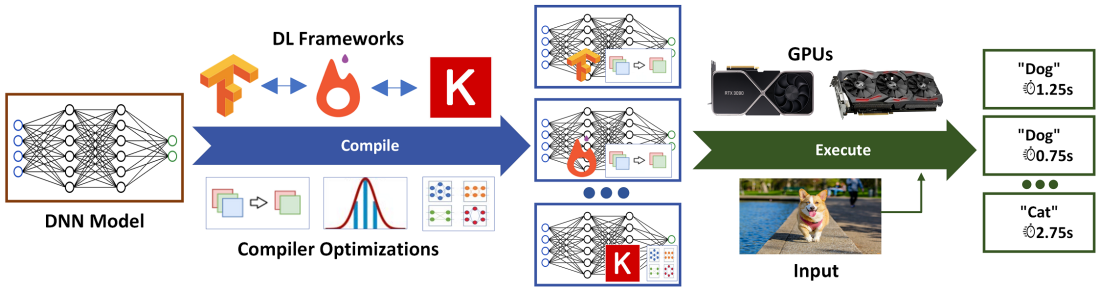


Figure 3.2: Differential Testing applied by `DeltaNN` for a DNN model, varying deep learning frameworks, compiler optimizations, and hardware devices.

Hardware devices: we generate implementations for a range of GPU accelerators, from a resource constrained mobile GPU to a powerful server-class GPU [218]. We consider different types of devices to check if GPU specifications can impact model output.

We assess the robustness of the DNN models by measuring changes in the output label classification predictions of a model across all three computational environment parameters, using images from a standard dataset as model inputs. Additionally, we monitor model inference times for different settings of computational environment parameters to understand the extent of variation among them distinguishing the cases where differences in model inference times are expected (e.g., across GPUs with different capabilities) or unexpected (e.g., the same model compared across DL frameworks). Inference times are a critical factor in ensuring timing safety for real-time perception systems deployed in safety-critical applications such as self-driving cars. These systems must operate within strict temporal constraints due to their safety-critical nature [219].

We assess the robustness of three widely used image recognition models – MobileNetV2 [71], ResNet101V2 [67], and InceptionV3 [72], on the ImageNet object detection test dataset (ILSVRC2017) [69]. We chose the three models based on their popularity (the papers related to these model versions have a total of ≈ 82K citations, and they rely on contributions of ≈ 366K citations in total [66, 220, 221]), but also to provide variety on layer architecture and model size (ranging from $48-347$ layers). The dataset we selected is a competition test dataset designed to extensively benchmark models. The `DeltaNN` framework uses the Apache TVM [60] machine learning compiler stack, as it allows importing models from all major DL frameworks while providing fine-grained control over compilation configurations and execution of the DNN models, as well as a wide range of hardware backend support.

Overall, we find that conversions between DL frameworks significantly impacts output labels of the DNN models. In particular, we compared the top-1 label for each sample in ILSVRC2017 across all models under test and observed differences in the predictions of up to 100% across the original and the converted model, (i.e., all ILSVRC2017 samples under test resulted in different top-1 prediction outputs). We identify that small amounts of noise introduced in the weights by the framework conversion tools are responsible for these issues, as a result of problematic conversion. The weight differences, although small, may be caused by differences in floating-point precision which can accumulate and cause label changes across the layers. On the other hand, we found that deploying models on different hardware accelerators and applying compiler optimizations do not affect model output accuracy but can lead to a non-negligible inference time performance degradation under specific scenarios, as we did not observe any top-1 output label prediction differences when varying these aspects. We observed up to 81% performance degradation in model inference times when applying certain compiler optimizations on some low-end hardware acceleration devices, which was unexpected, given that the same optimizations resulted in faster inference times on other devices with higher computational capabilities.
In summary, we make the following contributions:

1. Assess robustness of image recognition model *outputs*, post training, with respect to changes in the computational environment: DL frameworks, compiler optimizations, and hardware devices using a differential testing framework, `DeltaNN`.

2. Assess robustness of *model inference time* with respect to changes in the computational environment: DL frameworks, compiler optimizations, and hardware devices. While some measurements can be approximated by the hardware capabilities, we conduct experiments and compare the results across these aspects per-model, to observe how they behave in practice across the models under test.

3. Analyze and identify potential sources of *label discrepancy* when converting between DL frameworks.

3.2 Methodology

DeltaNN comprises three stages, as shown in Figure 3.3: (1) *Model Variant Generation* that generates different equivalent model implementations when changing DL frameworks and compiler optimizations; (2) *Differential Execution* that executes each of the model implementations with images from a test dataset; and (3) *Analysis* that compares the output labels, inference time, and other data from the different implementations, and aids in localization of discrepancy sources, if any.

3.2.1 Model Variant Generation

As seen on the left-hand side of Figure 3.3, the **Model Variant Generation** stage takes as input a pre-trained image recognition model (e.g., InceptionV3) sourced from a given DL framework (e.g., PyTorch). If we use one of these pre-trained models "as-is", and pass it directly to the **Model Importer**, we refer to it as a *native* model. However, if we convert it using the **DL Framework converter**, we refer to the original model as the *source*, and the converted model as the *target*. For example, we could convert an InceptionV3 model sourced from PyTorch to the TensorFlow model format. Across the four DL frameworks we support in DeltaNN, we have conversion paths from every framework to every other one.

To implement the conversions, we use the popular ONNX format [58] as an intermediate representation when a direct conversion is not possible. Some DL frameworks, such as PyTorch and TFLite, have native tools for this conversion; whereas for others, such as TensorFlow, we leverage popular third-party conversion tools like `tf2onnx` [54]. We then convert from ONNX to the target DL framework model format using a number of widely used libraries, such as `onnx2torch` [57], `onnx2keras` [56].

Compiler Configuration DeltaNN generates model implementations with different levels of TVM graph-level compiler optimizations: basic, default, and extended variants. **Basic** (o0) applies only "inference simplification", which generates simplified expressions which result to a model graph semantically equivalent to the original DNN. **Default** (o2) applies all optimizations of o0, as well as operator fusion for operations such as ReLU activation functions, as well as constant and scale axis folding. The optimizations are applied to the Relay intermediate representation (IR) [222] of TVM. **Extended** optimization (o4) applies all aforementioned optimizations, as well as

additional ones such as eliminating common subexpressions, applying canonicalization of operations, combining parallel convolutions, dense matrix and batch matrix multiplication operations, and enabling "fast math" (which allows the compiler to break strict IEEE standard [59] compliance for float operations if it could improve performance). We can also enable and disable specific optimizations at a fine-grained level, which can be useful for fault localization. In addition, kernel-level optimizations such as schedules, auto-tuning [212], third-party libraries (such as cuDNN [214]), and auto-scheduling [213, 223] can be explored, but are not the focus of this study.

Code Generation The final part of the *Model Variant Generation* stage takes the selected compiler configuration and imported model format and generates both host and device code, with the option to explore different programming paradigms (e.g., OpenCL [49] and CUDA [50]), CPU-side optimization flags (e.g., enabling vector instructions), and hardware devices (e.g., different GPU devices). The code generation step produces the outputs of the whole **Model Variant Generation** stage, namely several model variants, each with a different setting for compiler optimization, DNN model source or target, and host/device code configuration.

3.2.2 Differential Execution

The next stage of `DeltaNN` is **Differential Execution** of the model variants from the previous stage. It consists of three main steps: (1) the **Preprocessing** module responsible for normalizing inputs for better model performance (with a variety of preprocessing functions to choose from); (2) the **Regular Execution** module that executes the model for different target devices; and (3) the **Debug Execution** module, that executes the model similar to the *Regular Execution* module but additionally generates execution profiling information that can be used for deeper performance and error insights.

Preprocessing It is a common practice to preprocess the inputs from the dataset before inference, similar to training. Examples of preprocessing include image resizing, input image pixels normalization, and more. By default, the module preprocesses the inputs based on the model architecture and source DL framework.

Regular Execution This module executes the model on a specified target device to perform inference against a specific input and generate an output prediction. Execution encompasses model loading, setting up execution parameters from configuration, and experiment management (i.e., multiple runs). The output from this module is **Model metrics**, namely label accuracies and inference times for each image executed on the model. This module orchestrates and performs model execution in bulk, executing inference of a whole dataset against the numerous module variants generated by the **Model Variant Generation** module.

Debug Execution As the main purpose of `DeltaNN` is differential testing, generating execution-based metadata is vital for analyzing possible sources of error in the model variants. The **Debug Execution** module performs model execution similar to the **Regular Execution** module. Nevertheless, during execution, the module generates profiling metadata and debug metrics associated with the inference process, such as tensor outputs of each layer, per-layer inference time, and hardware counters. This information is passed on as *debug traces* to the **Analysis** stage for fault localization. We use this module to visualize activation differences across layers in case output prediction differences are detected, as described in Section 3.2.3.

3.2.3 Analysis

For every pair of model variants, the **Analysis** stage compares labels and inference time from *Model metrics* for all images in the dataset. To compare labels, we compare the top-ranked predictions between the model variants. When an image generates different label predictions between a pair of model variants, the **Analysis** module compares the *debug traces* from the model variants inspecting differences in *per-layer activations*, *weights*, and the *graph structure*. For *per-layer activations*, we compare mean, max, and standard deviations statistics of the layer activations between the pairs of models. The **Analysis** module also provides the capability to visualize the differences observed in layer activations and weights.

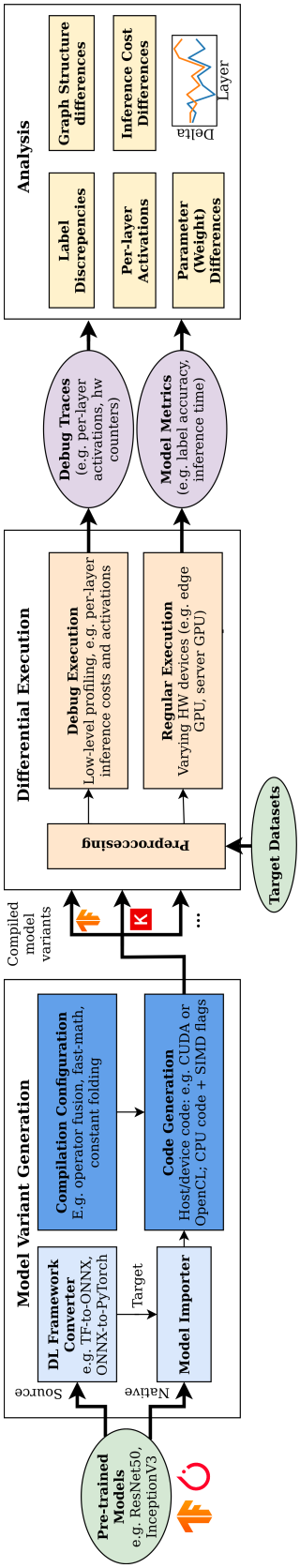


Figure 3.3: Architecture of the `DeltaNN` framework: (1) *Model Variant Generation* stage generates different model implementations when changing and converting DL frameworks and compiler optimizations; (2) *Differential Execution* stage executes the various model implementations on images from a dataset under test utilized for that purpose; and (3) *Analysis* stage compares output labels and inference time between executions while analyzing source of discrepancy.

3.3 Experiments

We consider three widely used image recognition models of various sizes: MobileNetV2 [71], ResNet101V2 [66, 67], and InceptionV3 [72]. We use models pre-trained on ImageNet [68], using native model definitions and pre-trained parameters/weights sourced from 4 different DL framework repositories: *Keras* [47], *PyTorch* [46], *TensorFlow(TF)* [51], and *TFLite* [51]. Each model is run through `DeltaNN` to generate model variants with different compiler optimization levels and *target* DL frameworks, and executed on 4 GPU devices (discussed in Section 3.3.2). In total, we evaluate a combination of 3 models, 12 DL framework conversions, 4 devices, and 3 optimization levels.

3.3.1 Research Questions

Our experiments are aimed at evaluating: (1) Robustness of model output, by recording the top-1 output label for every combination of environment parameters and performing pairwise comparisons; and (2) Robustness of model execution time, by measuring average inference time across executions in our dataset and comparing across different configurations. We investigate the following research questions for evaluating robustness:

Output Label Robustness

RQ1. Label Sensitivity to DL Framework Conversions *Are the output labels of an image recognition model affected when converting the model from a source to a target DL framework?* Both source and target frameworks are one among PyTorch, TF, TFLite, or Keras with $source \neq target$. All conversions are through the intermediate ONNX format. We compare output labels of target against the source for each image to check if any errors were introduced by model conversion. We do this for each of the three image recognition models – MobileNetV2, ResNet101V2, and InceptionV3.

RQ2. Label Sensitivity to Compiler Optimizations *To which extent are the output labels of an image recognition model affected when changing the level of compiler optimization?* We vary the optimization level within TVM between Basic, Default, and Extended and observe if there are any difference in the output label for images in the dataset.

Inference Time Robustness

RQ3. Time Sensitivity to DL Framework Conversions *Are the inference times of an image recognition model affected when converting the model from a source to a target DL framework?*

RQ4. Time Sensitivity to Compiler Optimizations *Are the inference times of an image recognition model affected when changing the level of compiler optimization?* We are aware that differences in inference time is to be expected to some extent when changing compiler optimizations. The goal here is to identify unexpected performance degradation and extent of change with the different compiler optimization levels.

3.3.2 Devices

We used four different GPU devices, featuring high-end to low-end accelerators:

- an Intel-based server featuring an Nvidia Tesla K40c (GK11BGL) GPU (*Server*), with a computational power of 4.29-5.04 TFLOPS for FP32 computations [224, 225],

- a Nvidia AGX Xavier featuring an Nvidia Volta GPU (*Xavier*), with a theoretical processing power of 32 TOPS and a calculated, theoretical performance of ≈ 1.4 TFLOPS [226],

- a Laptop with an i5-8365U CPU @ 1.6GHZx8, featuring an integrated Intel(R) GEN9 HD Graphics NEO (*Local*) with 24 Execution Units, and therefore a performance of ≈ 307 GFLOPS for FP32 operations [227],

- and a mobile-class Hikey 970 board featuring an ARM Mali-G72 GPU (*Hikey*), with a theoretical processing power of 244.8 GFLOPS for FP32 operations [228].

As a result, we categorize the Tesla GPU as our most powerful device, the NVidia Volta GPU as the mid-level device focused for AI tasks, and the other two devices as low-end.

For all GPU devices, we generate OpenCL device code—except for the Xavier device, which requires CUDA due to lack of OpenCL support. We observed no impact on output label prediction accuracy between OpenCL and CUDA, and the trade-offs between them have been previously studied [229]. We run the test dataset through the model configurations, and take the average inference time.

3.3.3 Dataset

The dataset used for effective testing must consist of real-world data, support meaningful model benchmarking, and remain tractable. For that matter, we use the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) object detection test dataset [69] in our experiments, consisting of 5500 RGB images that are generally resized to 224×224 pixels, and perform classification of 1000 possible labels and measure inference time on each image. ILSVRC is a well-known dataset, constructed with the purpose to evaluate models both for classification and object detection. It contains real-world, natural images which allow for the usage of the models in real-life scenarios. It has served as the benchmark for evaluating many major deep learning models, such as AlexNet [230], GoogleNet [231], and ResNet [66], while its size makes it suitable for tractable experimentation. Furthermore, as our experiments focus on classification models, we considered ILSVRC as a logical and appropriate dataset choice. Specifically, we used the 2017 version, which was the last one released. For

models native to TensorFlow and TFLite, we observed that models actually used input dimension of 299×299, rather than the typical 244×244 used by the rest of the model variants. In general, using larger input sizes could increase the potential accuracy of the models, but also the computational requirements they need to perform.

3.3.4 Execution Issues

All environment parameter combinations could not be executed with all models due to the following incompatibility issues. First, for ResNet101 sourced from PyTorch, we selected the V1 version the model instead of V2 as the V2 version was not provided in the official PyTorch repository. The version difference may have a larger effect on model inference time when we compare across DL frameworks. Second, regarding MobileNetV2, we experienced problems when executing it on the Xavier device, as we received a `CUDA_ERROR_INVALID_PTX` error, in all cases except when natively sourced from PyTorch. Thus, we do not consider this device configuration for MobileNetV2 in our experiments. Third, while utilizing the conversion process, we encountered various cases where the source model conversion failed, as presented in Figure 3.4. This happened due to incompatibility either of the source model with the conversion tool or the generated model operations in TVM.

3.4 Results

We present results with respect to discrepancies observed in output labels and inference time over the different DNN model configurations in the context of the research questions presented in Section 3.3.

3.4.1 Robustness of Output Label Prediction

RQ1: Label Sensitivity to DL Framework Conversions

The results are presented in Figure 3.4, showing the degree of dissimilarity between source and target models. As can be seen from the empty gray boxes, the conversion tool crashes in 11 out of the 36 conversions across the three DNN models, indicating that the conversion process failed. This happened due to compatibility issues between the conversion tool and a given model architecture, or the source or target DL framework. We repeated the conversion process for all models 3 times to verify the results, in order to minimize the possibility of randomness. The results were identical

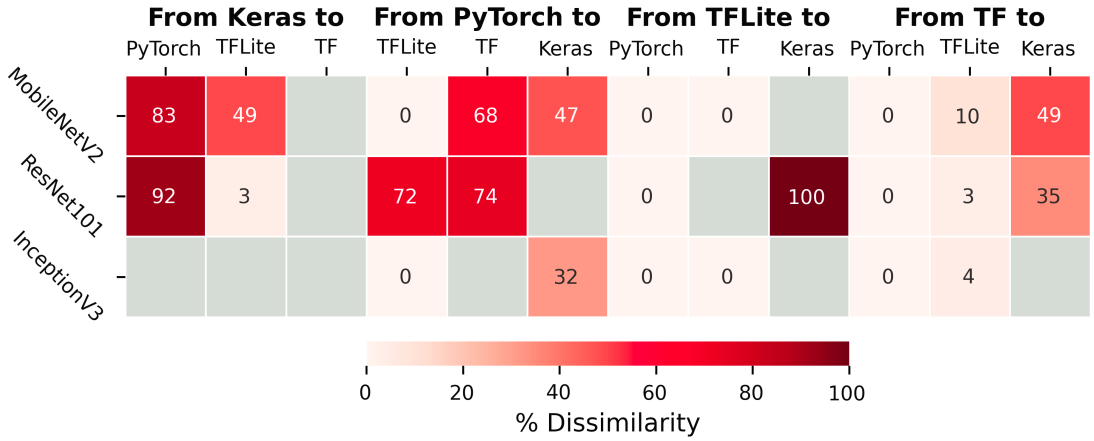


Figure 3.4: Pairwise comparison of output labels between *source* and *target* for a given model architecture across all images in the dataset.

on each run. For instance, we could not convert any Keras models to TensorFlow, due to the `tf2onnx` tool being unable to handle some tensor element values. Additionally, we observe further 15 cases where the conversion succeed without crashing, but the target model gave different labels from the source model, with 11 of these cases having a significant discrepancy (over 35%). In particular, we observe a 100% discrepancy in the output labels when converting the ResNet101 model from TFLite to Keras.

For conversions between either TF or TFLite and PyTorch, we observe no errors introduced by the conversion process across all models, while when converting TF to TFLite, we see relatively small discrepancies, 3-10%, demonstrating more reliable conversion. For TFLite to TF we had no discrepancies, but had one conversion failure (ResNet101). The difference in results compared to conversions in the opposite direction is due to the use of different converters, as the conversion process is not bidirectional. Each converter typically supports only one direction (for example, `TFLiteConverter` handles TensorFlow to TFLite conversions, but not TFLite to TensorFlow). This relative success is reasonable to expect, since TFLite has overlap with the TensorFlow codebase. However, ideally the differences should all be 0%. Table 2.1 shows that the native accuracy of the TensorFlow and TFLite models are all identical, implying that (1) the models are the same, and thus (2) the TFLite authors had 100% success with their conversions. However, we observe divergences using common open source conversion tools with default configurations.

Finally, the conversion of TF models to Keras gives varying results across models, with MobileNetV2 having 49% dissimilarity, ResNet101 having 35%, and InceptionV3 giving a model crash. This points to weaknesses in the conversion tool with certain DNN model architectures.

Fault Analysis We use the **Analysis** module of `DeltaNN` to explore in greater detail the cause of discrepancies across DNN model conversions. To illustrate the analysis in depth, we select one of the models and conversions that results in a discrepancy, InceptionV3 with TF as the source DL framework converted to target TFLite. The discrepancies observed across *source-target* is 4%, and in Figure 3.5 we show the class breakdown of the images which demonstrated differences, sorted by what proportion of that class showed discrepancies. We highlight a subset of the class labels on the x-axis. We observe that some classes are impacted more than others, with some classes such as "walker hound" disagreeing on 100% of the images. However, this graph is not indicating test set accuracy, instead it is about agreement between the *source* and converted *target* models, which under ideal circumstances we would expect to have equal agreement in all cases.

We also performed inference using the intermediate ONNX format, which is used as part of the conversion and model loading process to TVM, using an indicative selection of images from the ILSVRC dataset. We used *ONNXRuntime* [232] for that purpose. For all selected images, and we found that the TF model differed from the TFLite inference results, whereas the TFLite model results were identical to ONNX, as seen in Table 3.1 for five images as an example. The ground truth for these images matches the results from the source model, and deviates from TFLite and ONNX. This narrows down the source of the error mainly to the conversion tool from the source TF model to TFLite, i.e., `TFLiteConverter` [233] of the native TFLite API, and less to `tf2onnx` [54], which is widely used in the community (1.8k stars on GitHub). We store the tensor outputs from the source and target models for further analysis.

Next, we performed execution on the *source* and the *target* models using the `Debug Execution` module of `DeltaNN`, which relies on the debugger of TVM and provides metadata about the execution. Following this process, we perform per-layer activation analysis combining the debugger metadata with metadata of the build process for the *source* and the *target* models. We compare the average differences between the models across layers utilizing parameters (i.e., weights and biases from the convolution layers), per-layer tensor outputs (i.e., activations), as well as the

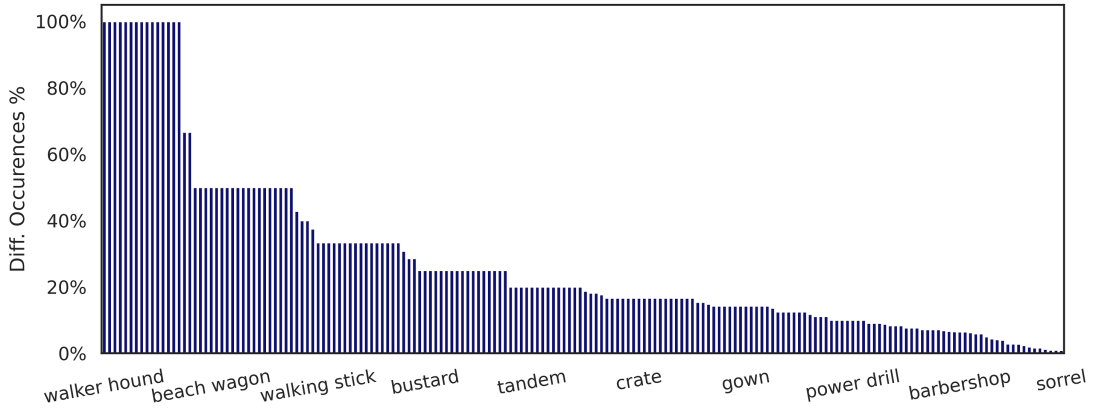


Figure 3.5: Percentage of affected images due to library conversions, InceptionV3, TF-to-TFLite conversion.

Table 3.1: Inference (top-1 prediction) of 5 ImageNet images posing different results between models using TF, TFLite converted from TF, and complementary ONNX applied on InceptionV3 and run on Local device using Default optimization.

Image ID	TF	TFLite (TF)	ONNX
00001219	scooter	moped	moped
00002078	cottontail	llama	llama
00002439	wallet	purse	purse
00003928	black grouse	bee	bee
00004898	wallaby	It. greyhound	It. greyhound

hyperparameter values for the respective layers. We illustrate this for two images in Figure 3.6, focusing on the convolutional layers, where *Image 1* generated the same output label across source and target models, whereas *Image 2* produced completely different labels across source and target models for the top-5 predictions. We observe that both images have divergences in their activations, but for *Image 2* the divergences are higher for later layers (layer 170 onwards).

Per-Layer Activation and Model Parameter Analysis Figure 3.6 highlights the difference between intermediate activation maps (i.e., the outputs of individual layers during execution), as well as the differences in the parameters. We would expect the models to behave the same, since they should consist of the same model architecture and parameters. Our observation is however that the output labels are not always consistent. For *Image 1*, both *source* and *target* versions of the model produce the correct label, even though their intermediate activations are between 0.0 and 0.06 on

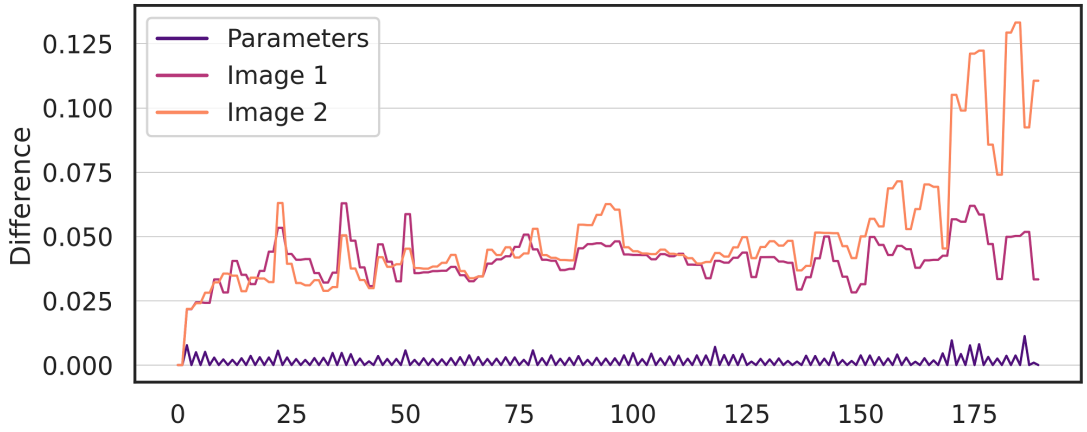


Figure 3.6: Layer-wise evaluation of the differences between a model sourced from TensorFlow, and converted to TFLite. "Parameters" shows the mean difference between their weights and biases. *'Image 1'* and *'Image 2'* show models' differences in activations for two inputs.

average. However, for *Image 2* the models disagree on the output label, and for later layers, we observe a higher average difference, up to around 0.13. This may imply that whatever error has occurred may have impacted later layers more, but this does not explain why this is not the case for *Image 1*.

To identify the source of these discrepancies, we examine the differences in the parameters of the models (seen as the green line in Figure 3.6), which indicates a possible source of the error. Across parameters, we observe a divergence of 0.0003 on average and 0.011 at most. In principle, when we run our conversion tool the parameters should be unchanged, i.e., bit-wise identical from TF to TFLite. Furthermore, we presume that the model parameters are incorrectly copied at some stage of the DL framework model conversion process. With this bug identified and fixed in the conversion tool, we could expect that the difference between the models goes away. We demonstrate this by replacing the parameters of the converted model with the source model within TVM, and observe that 100% of our divergence disappears. As an additional fault mitigation measure, we could also integrate the impact of the error into the model during training by simulating the conversion tool noise into our parameters, so that the model learns to be robust against it.

Therefore, the confirmation of our hypothesis still does not explain why we observed non-uniform divergence in the activation maps and output labels; despite the fact that the parameters had small, relatively uniform noise, and were identical between *Image 1* and *Image 2*. However, if we reason about the underlying operations

and mechanics of a DNN model, we can begin to make sense of it. We observe that the impact of these weight errors are cumulative, since weights are generally used repeatedly in multiply-accumulate operations. Then, layers with more such operations (e.g., ReLU activation functions) will be more likely to have higher errors.

By complementary examination of the conversion tools involved, we identify that `tf2onnx` did not introduce any weight deviations, but that `TFLiteConverter` was responsible for introducing the weight deviations. However, upon further inspection, we determined that these deviations are a result of arbitrary quantization set by default from the TFLiteConverter without any explicit warning raised for that, combined with implicit mishandling of the model by an additional conversion step from TFLite to ONNX applied again by `tf2onnx`, in order for the model to be loaded to TVM to perform per-layer activation and model parameter analysis.

RQ2: Label Sensitivity to Compiler Optimizations

We conducted experiments across all DL frameworks and device combinations described in Section 3.3, using only the native DNN model definition and varying the optimization level (Basic, Default and Extended), in order to observe inference time and output label discrepancies. Similarly to RQ1, we repeated the model optimization process for all models 3 times to verify the consistency of the results. We found them to be identical on each run.

We found that varying compiler optimization levels causes no discrepancies in output labels for all three models. The lack of discrepancies or sensitivity is notable, as the Extended (-o4) optimization level enables unsafe math optimizations that can generate code violating the IEEE-754 floating-point arithmetic standard. The conclusion is that these potential unsafe perturbations were so small that all three models were resilient to them. In addition, we focus only on the top-1 label output label prediction for accuracy deviations, as it represents the most important prediction in classification models. However, such perturbations may still impact some or all the remaining label predictions. It is also not a foregone conclusion that the ostensibly semantic preserving optimizations of Basic and Default optimization levels would have produced no label divergences, as compilers are complex systems that might contain bugs. However, TVM's optimizations did not introduce any errors in our experiments.

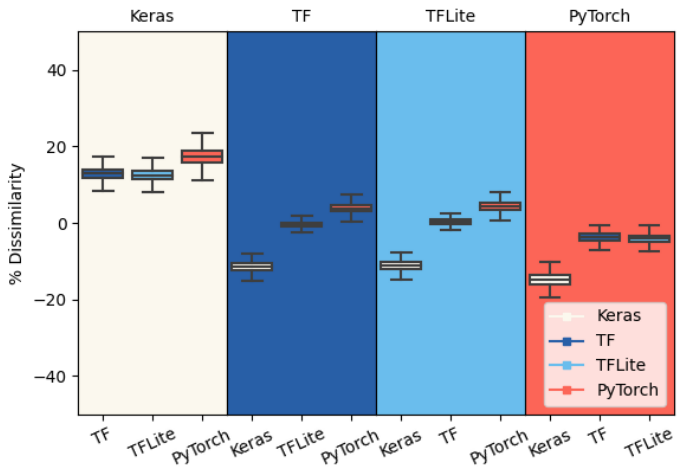


Figure 3.7: Inference time differences (%) between DL frameworks on Server, for MobileNetV2, with Default Optimization.

3.4.2 Robustness of Model Inference Time

RQ3: Time Sensitivity to DL Frameworks

We observed a 4–16% difference in inference time using the MobileNetV2 model with Default optimization across deep learning frameworks deployed on the Server, with the largest difference of 16% occurring between Keras and PyTorch, as shown in Figure 3.7. The differences were confirmed to be significant using one-way ANOVA with 5% significance level. We believe the difference is due to the different graph representation after framework conversion, e.g., one framework may represent a fully-connected layer as a "dense" operation and another may represent it as a "batch matmul"; or some conversion tools may apply some of the graph-level optimizations such as batch-normalization fusion, so that even with a Basic optimization level the model is simplified.

RQ4: Time Sensitivity to Compiler Optimizations

We observed a positive relative change of up to 121% in inference time with increasing optimization levels (when applying Extended optimization on ResNet101 from TFLite, and deploying on Hikey). As part of our statistical analysis, we confirmed the observation to be significant using one-way ANOVA with 5% significance level. This is not surprising, as different optimization settings have a direct effect on code efficiency.

Interestingly however, there were instances where increased optimization led to a slowdown in inference time. For instance, MobileNetV2 from Keras and Extended optimization presented a -81% relative change compared to Basic on the Hikey device (Figure 3.8). We also confirmed our observations using one-way ANOVA.

To explore the impact of the compiler in greater detail, we enabled optimization passes individually, taking each optimization concept and applying it to the model separately, rather than using multiple optimizations together as the `-oX` bundles that we use at a high level. We conducted an analysis on 100 images, using one optimization pass per-case in isolation, to understand which optimizations contributed to speedup or slowdown in model inference time for ResNet101 and InceptionV3 using TensorFlow and PyTorch DL frameworks[1].

We found that no single optimization led to a significant inference time change for this experiment, suggesting that the non-trivial interactions between optimization passes are what contribute to these changes, making model optimization more challenging. For ResNet101, the optimization which provides convolution operators' scale axis folding *negatively affected* the performance by 2.71% on the Local device, while the combination of parallel operators had a positive impact of up to 2.82% compared to using Basic optimizations alone. With InceptionV3, constant folding had a positive impact of 4.9%, while combining parallel operators *negatively affected* the performance by up to 5.47% compared to Basic optimizations alone. The difference in effect of optimizations between InceptionV3 and ResNet101 is likely due to the difference in their model architecture and data flows.

However, we observed that the combination of optimization strategies can lead to significant performance degradation under certain contexts, such as low-end hardware acceleration devices. Figure 3.8 shows the relative percentage change in inference time for Basic versus Extended optimization on different devices and models with PyTorch, as an indicative example, using the Extended optimization as the baseline. For each device, we find that times generally improve with increased optimization in the range of 3.8-8.4% for Server, and 17-54% for Local. Additional optimizations on Hikey, however, had a 81.8% negative relative change (confirmed with One-way ANOVA 5%). The Xavier device also had a 36% negative relative change (confirmed with One-way ANOVA 5%) when increasing the optimization level from Basic to Extended on InceptionV3 model. For low to mid-range devices, Xavier and Hikey, we experienced a slowdown with increased optimization. We hypothesize that the limited

[1]Our results can be found at: `https://github.com/luludak/deltann-results`

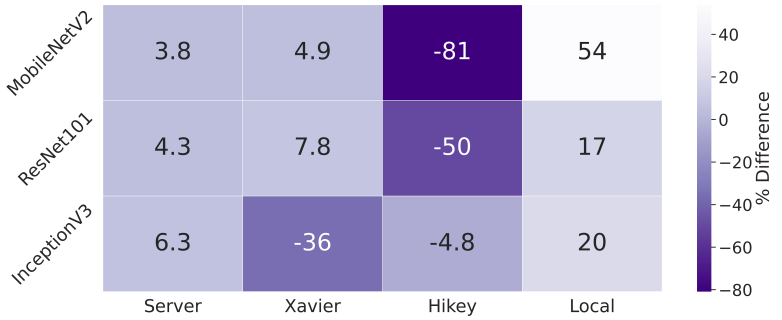


Figure 3.8: Relative changes of inference times (%) between Basic and Extended optimizations across all devices, for all PyTorch models, compiled and deployed using Apache TVM

.

GPU memory poses a problem for the optimizations with parallel operations in the Extended optimization setting, leading to additional wait times, context switches, and GPU data transfer time, which result in a slowdown. Investigation of cache behavior, data transfer times between the CPU and GPU, and processor idle times to clearly identify the reasons for slowdown can be an interesting extension, but is currenty outside the scope of this work.

3.4.3 Threats To Validity

There are five main threats to validity in our experiments:

1. We only evaluate robustness using three image recognition models that are widely used (the papers associated with them have over 330 thousand citations in total). The results are model dependent as seen in our experiments and will likely vary on other models; however, given that the models we selected are popular among developers and researchers and vary in parameter size and capabilities, we considered them good candidates for our experiments.

2. We use the ImageNet [69] object detection test dataset for our experiments, which we believe adequately stresses configurations, as it is widely used to benchmark widely used models, such as ResNet [66]. We are aware however that other datasets may yield different robustness results on the models considered;

3. Model preprocessing is crucial for model performance [234], and models may give suboptimal performance if given data with ineffective and erroneous preprocessing. We use the recommended preprocessing for each model and DL framework from the official repositories extracted;

4. Beyond the DL framework conversions explored in our results, we also have a "hidden" conversion step, i.e., importing models into Apache TVM, which itself may introduce errors. To ensure that errors are not introduced before loading each model into TVM, we generate "target outputs" from their source framework using an indicative number of random image samples. After importing into TVM, we confirmed that we match the target outputs, however this may not guarantee that the import process is entirely bug-free;

5. We consider the potential deviations of both accuracy and inference time measurement. We repeated model conversions and optimizations 3 times and verified the results remained consistent across all runs. In addition, to ensure that potential time deviations are taken into account, we repeat inferences 10 times for each image and use the average inference time across each run across a small-scale test dataset, verifying that no deviations happen on scaling. Although we cannot guarantee result consistency with 100% confidence, our checks increase our confidence in the results. Note that non-trivial medium-term cache behavior may cause inference time to vary over repeated runs, and depending on the deployment scenario of interest, only the "first" inference time may be of interest, or the inference time of the 'Nth' sample where N is a large value.

3.5 Lessons Learned

Our empirical study exploring the effect of changing computational environment parameters revealed the following findings:

Failures in DL Framework Conversion: Automated conversion of models between DL frameworks can introduce significant output label discrepancies. We observed up to 100% output label dissimilarities when converting TF Lite to Keras for ResNet101. In addition, converting Keras to TF generated failures for all three models under test. These errors can be introduced in model weights, parameters, graph and architecture representation during the conversion process. Our analysis revealed errors in model weights introduced by the converters when converting from the source model (TF) to TFLite, which can be fixed by correctly copying over the source model weights.

DL Framework Conversion - Impact on Model Inference Time: Changing the DL framework used to generate the model can have a considerable effect on model inference time. The extent of this impact depends on other environment parameters. Inference time impact varied from $4 - 16\%$ with the largest impact (16%) observed between Keras and PyTorch (Figure 3.7).

Performance Degradation from Compiler Optimizations: Compiler optimizations are generally expected to improve the performance of a model. However, our findings indicate that for certain scenarios defined by the device, model and library, compiler optimizations can be detrimental to model inference time. In our experiments, we observed this performance degradation to the greatest extent when applying Extended optimization to MobileNetV2 for the Hikey device, which resulted in 81% performance degradation when compared to a lower optimization level using Basic.

In a Nutshell: Overall, we recommend that DNN model developers and researchers conduct comprehensive testing prior to deployment in production environments. Models should be evaluated under conditions that closely match the target production configuration, including the deep learning frameworks used, any model conversions or compiler optimizations applied, and the specific hardware devices targeted, with attention to their computational characteristics. Crucially, testing should incorporate real-world datasets (e.g., ILSVRC) and assess both accuracy and execution time metrics. Finally, it is important to note that in safety-critical applications, the consequences of the above sensitivities can be crucial. Therefore, it is essential that framework, compiler, and hardware communities, along with the developers of DNN models are aware of these sources of error, and test their systems for robustness to computational environment changes. Currently, there is no regulation or benchmarking of DNN model performance and accuracy for environment parameter configurations. The results from our study indicate that assessing sensitivity to environment parameters is an important consideration during model development and use. Model developers should test DNN models under the actual configurations used in production environments to verify their intended performance - both in terms of accuracy and inference times.

3.6 Summary

We introduced DeltaNN, a differential testing framework to explore the impact of computational environment parameters on image recognition models. DeltaNN allows DNN model compilation, optimization and deployment across different hardware acceleration devices, as well as model inference across a user-defined dataset and comparison of the results across configurations. In our work, we study the effect of converting between popular deep learning frameworks (TensorFlow, Keras, TFLite, PyTorch), compiler optimization settings, and hardware devices on the output labels of three widely used image recognition models (MobileNetV2, ResNet101, InceptionV3). We monitor the impact of these parameter effects towards two metrics, (1) model output label correctness and (2) prediction inference time.

Overall, we observed that model conversions between DL frameworks was found to be prone to errors, with 11 out of 36 cases under test to crash completely, while the conversion process led to degradation of the model output label prediction correctness, in 15 more cases, with models presenting up to 100% output label prediction differences across the converted target model and its original source version. We also found that compiler optimization effects towards DNN models might have varying inference time performance effects based on the device capabilities. For instance, extended optimizations had 3.8% inference time improvement for a high-end GPU (Nvidia Tesla K40), while the same optimization setting led to severe performance degradation (81% penalty in inference time) to a low-end GPU (Mali-G71) in comparison to the configuration not containing any model optimizations.

3.7 Availability

The source code of DeltaNN, accompanied with detailed instructions of installation and usage, are available at: https://github.com/luludak/DeltaNN

Chapter 4

Automatic Fault Localization and Repair of Deep Learning Model Conversions

4.1 Introduction

With the widespread use of deep learning (DL) in various domains, there is an inherent need for DL models to be interoperable across DL frameworks (such as PyTorch [46], TensorFlow (TF) [51], Keras [47]) to maximize re-usability of models across frameworks. Conversion of DL models between frameworks is facilitated by a plethora of automated conversion tools such as `tf2onnx` [54], `onnx2keras` [56], `onnx2torch` [57], `MMdnn` [53], among others. However, the conversion process may be riddled with bugs [70, 74, 235], making the converted models undeployable, perform poorly in terms of output label correctness when changing from one framework to another, run slowly, or face challenges in robust deployment [42, 83, 188].

In Chapter 3, we provided empirical evidence of model conversion errors and resulting output label errors in four DL frameworks. In this chapter, we target the conversion errors we observed for fault localization and repair. We considered three image recognition models pre-trained on ImageNet [68] that were treated as *Source* models. We then converted each *Source* model to use a different DL framework, referred to as *Target*. Figure 3.4 from [70] shows the proportion of output label dissimilarities between *(Source, Target)* pairs across all images in the test dataset. The conversion process failed to execute in 11 out of the 36 conversions, seen as empty gray cells in Figure 3.4. For the remaining conversions, we found varying levels of image label discrepancies between *Target* and *Source* models from 0 to

100%, with 15 cases having non-zero percentage of output label dissimilarity. This study underscores the error-prone nature of the conversion process, highlighting the necessity for a technique to localize and repair faults introduced by DL framework converters.

To address this problem, in this chapter we propose `FetaFix`, an automated approach for fault localization and repair of erroneous model conversions between DL frameworks. We focus on the error cases in Figure 3.4 with non-zero output label differences. The gray boxes in Figure 3.4, where the conversion tools crashed, have been explored by existing tools like MMdnn [53] and complements the capability we provide in diagnosing and repairing conversions with output label differences.

`FetaFix` is capable of detecting and fixing faults introduced in model layer weights, biases, hyperparameters, and the model computation graph during conversion. It compares the *Source* model (in one DL framework) against the converted *Target* model (in a different framework) using an input dataset. For images with label discrepancies between *Source* and *Target*, `FetaFix` assesses the difference between the models with respect to their parameters. Then it uses a set of strategies to mitigate conversion errors, such as the replacement of *Target* model parameters with those from *Source*.

Our approach for `FetaFix` focuses on the *Target* model rather than the converter tool itself for a variety of reasons. First, errors observed in conversions may be related to problematic configurations (e.g., input preprocessing) that are not related to the conversion tool. Second, the source code of the conversion tool may be unavailable or proprietary, and therefore not accessible for direct repair. Third, the conversion process might involve manual intervention or conversion (e.g., [236, 237]), which in turn can introduce errors. Fourth, our methodology is not tied to any specific conversion tool/version, and can be applied to any model conversion between source and target settings, including intermediate representations like ONNX [58]. It is worth noting that conversion tools are constantly evolving, thus having a localize/repair approach independent of the tool source code accessibility and evolution would prove helpful and more dependable for AI model users. An important counter-argument to applying repairs directly to the models instead of fixing the converter is that, once the conversion tool is corrected, the specific issue is permanently resolved and does not require repeated mitigation. While this is valid, we argue that our tool primarily targets the repair process rather than a single model. Although the repair is applied to the model under test, `FetaFix`systematically logs and highlights the components

of the model that contribute to accuracy degradation. These logs can, in turn, aid in isolating and identifying faults within the converter itself. Furthermore, the fault localization and repair techniques we propose can also be adopted by conversion tool developers to address faults associated with the tools themselves.

We evaluate the effectiveness of `FetaFix` in fixing the model conversion errors we observed on Chapter 3. `FetaFix` was able to effectively localize and repair 14 of the 15 erroneous model conversions that had non-zero output label dissimilarities.
In summary, we make the following contributions:

1. A novel method for localizing faults in DL model conversion between a range of DL frameworks.
2. An automated fault repair framework, `FetaFix`, that can fix converted *Target* models to match output labels with that of the original *Source* model.
3. A comprehensive empirical evaluation of the effectiveness of `FetaFix` in fault localization and repair.

4.2 Methodology

`FetaFix` first takes as input the dataset, *Source* and *Target* models. It then localizes and repairs conversion errors that resulted in output label discrepancies between *Source* and *Target*. To compare the *Source* and *Target* model architectures, configuration and parameters we use their corresponding representation in the ONNX format which is a popular intermediate representation. We then import the *Source*-ONNX and *Target*-ONNX models to the TVM compiler framework to facilitate comparison. `FetaFix` comprises two main components: 1) A Fault Localization component that identifies locations, parameters and properties in the *Target* model that may be potentially faulty; and 2) a Fault Repair component with a defined set of strategies to fix the faulty locations and aspects identified through fault localization. The full fault localization and repair pipeline of `FetaFix` is presented in Figure 4.1.

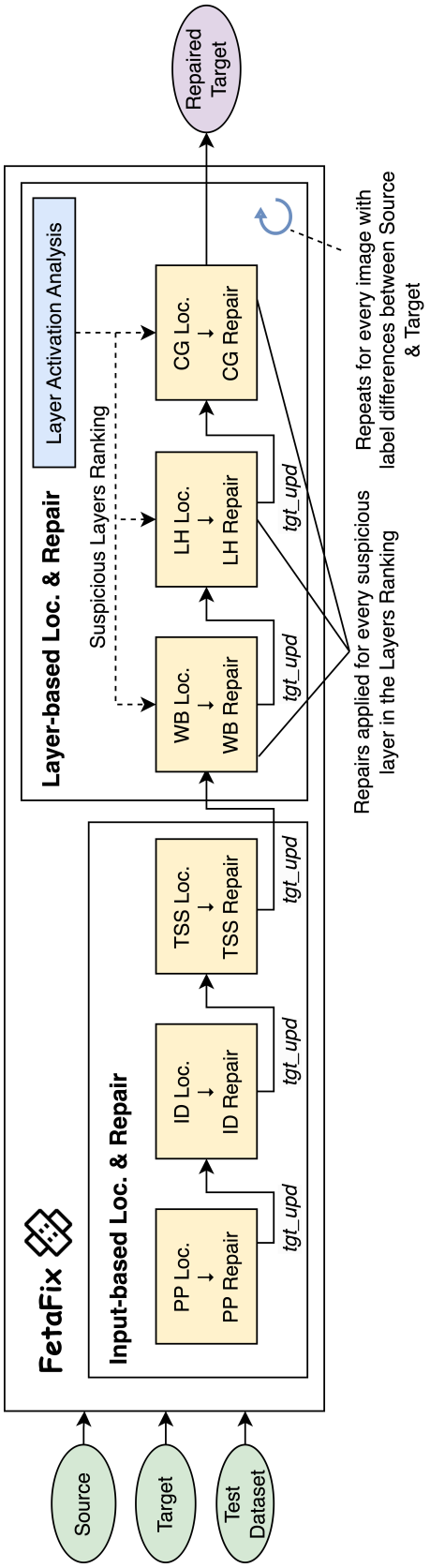


Figure 4.1: Fault localization and repair pipeline of `FetaFix` handling 6 fault types, three within Input-based category and three within Layer-based category.

4.2.1 Fault Localization

We surveyed 40 posts and issues from StackOverflow, GitHub, and forums such as TensorFlow and NVIDIA discussions, where model conversion errors were raised, to understand the possible locations of errors. Although not exhaustive, we are confident that this sample is representative of issues faced by developers, as (1) the issues we observed were among the most popular with regard to conversions posted in the Q&A websites and forums and (2) the same or very similar issues appeared across different trackers and forums. Table 4.1 shows six different conversion fault types identified in these issues that were related to 1) Preprocessing (PP), (2) Input Dimensions (ID), (3) Tensor Shape and Structure (TSS), (4) Weights & Biases (WB), (5) Layer Hyperparameters (LH), and (6) Computation Graph (CG). We also discovered many of these fault types examining the conversion errors reported in Chapter 3.

Our methodology is primarily inspired by conventional differential testing [104], adapted to the context of deep neural networks (DNNs). It directly compares the behavior of two comparable model variants — *Source* and *Target* — to identify and localize faults, following a greedy approach towards fault localization. Additionally, it draws inspiration from delta debugging [238], aiming to identify the minimal set of problematic attributes that lead to accuracy deviations in the *Target* model relative to *Source*. The process begins by applying fault localization to non-repetitive system operations, such as preprocessing. If no repair emerges at this stage, `FetaFix` proceeds with an iterative, search-based strategy that targets one layer-based category at a time. This approach seeks to isolate layers in *Target* that (1) differ from those in *Source*, and (2) impact model accuracy, using a divide-and-conquer technique.

To enhance this process, we incorporate a variant of Search-Based Fault Localization (SBFL) [239, 240]. Specifically, for layer-based faults, we analyze which layers are most likely to introduce accuracy-affecting differences between *Source* and *Target*. Unlike traditional applications of SBFL in code analysis, our approach is tailored to DNNs. We estimate the number of "suspicious" elements in a layer by comparing activations between *Source* and *Target* using a small set of inputs. After determining the number of suspicious elements in each layer, we rank the layers based on this information.

Similarly to fault localization, we combine one-off repairs with greedy Search-Based Fault Localization (SBFL) [239, 240] in order to repair the *Target* model. Using the layer ranking and fault localization information, `FetaFix` performs one-off repairs for the input-based strategies, and SBFL for the layer-based strategies, using

one sample input as the sample case that repairs will be tested against quickly. In case a repair is proven to be effective to this sample (improving its accuracy), then the potential fix will be tested against the whole dataset, and if found effective, it will be integrated to a new baseline *Target*. The process is repeated until *Target* is fully repaired, or `FetaFix` cannot apply any additional repairs.

Table 4.1: Model conversion faults from StackOverflow, GitHub issues and other forums. Faults are classified as input-based and layer-based.

#	Class	Fault	Sources
1	Input	PP	[241, 242, 243, 244, 245, 246]
2	Input	ID	[247, 248, 249, 250, 251, 252, 253, 254, 255]
3	Input	TSS	[256, 257, 258, 259, 260, 261]
4	Layer	WB	[262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273]
5	Layer	LH	[274, 275, 276]
6	Layer	CG	[277, 278, 279, 280]

As seen in Figure 4.1, our fault localization approach starts by localizing the first three fault types: PP, ID, and TSS. These three fault types pertain to the input, stemming from differences between *Source* and *Target* with respect to the preprocessing used, or changes in input dimensions or operations on the input. We refer to these fault types as Input-based. After attempting to localize input-based fault types, our approach proceeds with the remaining three fault types in Table 4.1 that belong to the Layer-based category, as it stems from differences in layer hyperparameters, weights and biases, and differences in computation graph.

Comparisons for Input-Based Fault Types

We examine and contrast input-related settings between the *Source* and *Target* models in the fault types below.

Preprocessing (PP)

For some instances, differences in preprocessing settings between *Source* and *Target* can result in output label differences. We explore the effect of different preprocessing configurations for *Target* using the default preprocessing setting from the official repositories of the DL frameworks both *Source* and *Target*. For example, in the case of converting MobileNetV2 from PyTorch to Keras, `FetaFix` examines the output label prediction accuracy of the converted *Target*, using both official PyTorch and Keras preprocessing settings for MobileNetV2.

Input Dimensions (ID)

Islam et al. [120] identified that changes in layer input dimensions can affect the performance of a DNN model. We also empirically identified that model conversion tools accept misconfigurations of *Target* model input dimensions without notifying the user of the mismatch between *Source* and *Target* input dimensions that may potentially result in output label discrepancies. We explore problems arising from such misconfigurations in this fault type. For example, we inspected the conversion of ResNet101 from PyTorch to TFLite, modifying the input dimensions from `(1, 3, 224, 224)` used in *Source* to `(1, 3, 299, 299)` used in *Target*. The conversion process completed without errors and warnings. However, the model presented a significant output label discrepancy between *Source* and *Target* (37%). In our fault localization approach, we check if input dimensions between *Source* and *Target* match, and report any mismatches to the fault repair component.

Tensor Shape and Structure (TSS)

Another potential error identified by Islam et al. [120] relates to tensor shapes and changes in model structure. We consider cases where an erroneous tensor shape can affect the correctness of the model conversion where *Source* and *Target* models have differences in output labels. Such a scenario may be observed when a transpose operation is introduced by a converter tool to modify the model input. This is typically manifested as a `Transpose` node right after the input node in the computation graph. In an erroneous conversion scenario, the `Transpose` node might generate a tensor with correct shape, but incorrect structure. To illustrate this, consider the example of a *Source* input with the shape `(1, 224, 224, 3)`. The *Target* model on the other hand has a shape of `(1, 3, 224, 224)` after applying a `Transpose` node that modifies the input of *Target*, by either applying `Transpose(tensor, [0, 2, 3, 1])`, or `Transpose(tensor, [0, 3, 2, 1])`, depending on the implementation. The modified shape is used in subsequent nodes in the computation graph. Owing to the ambiguity in dimensions where both transpose operations result in a tensor with correct shape expected by the subsequent layers in the *Target* model, it can result in model conversion errors as one of them has an incorrect structure. To localize faults like this, `FetaFix` checks if there is a difference in model input shape between *Source* and *Target*. For cases presenting differences, `FetaFix` further examines if any of the input-related subsequent layers contains a shape transformation operation (like `Transpose`) and considers such scenarios as potential sources of faults, conservatively.

Potential faults related to tensor structure may also happen when a layer tensor in the *Target* model is incorrectly transformed by the converter and is followed by a flattening or a reshaping operation that loses the original structure information, making the difference with *Source* model non-obvious. In particular, if a tensor has a different shape between *Source* and *Target* (i.e., due to erroneous conversion) but the total number of elements is the same across tensors, a `Flatten` or a `Reshape` operation on such a tensor will make it impossible to detect the mismatch between *Source* and *Target*, as the structure information is lost. However, the order of elements between the corresponding *Source* and *Target* tensors will be different after the `Flatten` or a `Reshape` operation, which may result in output label discrepancies. For instance, if a layer has shape `(1, 3, 32, 32)` on *Source* but was converted to `(1, 32, 32, 3)` on *Target*, and is followed by a `Flatten` node in both models, the result will be tensors with `(3072)` elements in both models. We localize this fault by examining the tensor shape of the dominator nodes preceding any `Flatten` or `Reshape` nodes, and check if it matches with the corresponding nodes in the *Source* model.

Comparisons for Layer-Based Fault Types

For this category of faults, we examine and compare layer activations between *Source* and *Target* models. For layers where differences in activations are outside of an expected range of differences (as described in Section 4.2.1), we mark those layers as suspicious and rank them so that we examine the most suspicious layer first for fault types CG, LH, and WB. We then iteratively proceed down this ranked list of suspicious layers for inspecting the three layer-based fault types, CG, LH and WB, discussed below. We start by describing our approach for producing a ranked list of suspicious layers in layer activation analysis followed by our fault localization approach for CG, LH, and WB with the ranked suspicious layers. An example instance of the differences between *Source* and *Target* in weights, biases, and hyperparameters that `FetaFix` attempts to localize as potential faults is shown in Figure 4.2. We find, for a core computational layer under inspection, that the weights present small differences between *Source* and *Target*. Additionally, the `pads` hyperparameter that is present in *Source*, is missing on *Target*, as demonstrated in Figure 4.2.

Layer Activation Analysis

We compare the activations for each convolutional layer between *Source* and *Target* models, as a four-step process:

- (1) We extract two equally sized sets of images from the test dataset: one for similar images (images that produce the same top label prediction across *Source* and *Target*), and one for dissimilar images (images with different labels between *Source* and *Target*). We then extract the layer activation tensors of both sets by loading the *Source* and *Target* ONNX models to TVM and performing inference using the *TVM Debugger* [281]. The TVM debugger generates the complete model graph and parameters at compile time, while also gathering layer activations for each input. Using these activations and the *Source* ONNX model, we then utilize the `layer_analysis` function, shown in Algorithm 1. This function takes as input the original source ONNX model (used to iterate the matching layer activations for both *Source* and *Target*), as well as the activations for the sets of similar and dissimilar images for both *Source*and *Target*(contained in parameters `sim_acts` and `diss_acts`). It also utilizes the `List` module, responsible for list length check and sorting operations, and the Kruskal-Wallis parametric test (`KWTest`).

- (2) To identify when activation mismatches between *Source* and *Target* indicate a potential problem, we first extract the differences in activations between *Source* and *Target*, per element within each layer, for the set of similar images and store these differences in a set, `sim_diff` in line 8 of Algorithm 1. We treat differences in activations between *Source* and *Target* for these similar images as acceptable differences and we use it to estimate the expected difference in distribution of activations for correct conversions.

- (3) Next, for the set of dissimilar images, we compute the set of dissimilar differences, `diss_diff` in line 10, per tensor element per layer. We then check if the difference in activations per tensor element per layer belongs to the corresponding expected distribution of activation differences (from the previous step). To do this, we use the *Kruskal-Wallis non-parametric statistical test* (line 11) [282][1] to determine if `sim_diff` and `diss_diff` belong to the same distribution (at 5% significance level) (line 15).

[1]The distribution of activation differences for correct conversion instances did not follow a normal distribution, so we chose the non-parametric Kruskal-Wallis test.

- (4) From step 3, we obtain the tensor elements whose difference in activations for `sim_diff` and `diss_diff` did not belong to the same distribution, for each layer between *Source* and *Target*. These are considered as potentially problematic elements that may result in model conversion errors. We count the number of problematic tensor elements for each model layer (line 16) and rank the layers in descending order of number of problematic elements (line 18). This ranking of suspicious layers is the basis for fault localization and repair of fault types CG, LH, and WB. We only consider convolutional layers as these are the core computational layers for a Convolutional Neural Network (CNN). However, this can be modified to consider other layer types as well.

An interesting extension to this analysis would be a causality-based approach, where activation values are slightly modified and the resulting changes in model behavior are observed. This could provide deeper insights into which layer activations are more causally responsible for potential issues in the system.

Algorithm 1 Layer Analysis in `FetaFix`

```
Require: List, KWTest, model, sim_acts, diss_acts
 1: procedure layer_analysis(model, sim_acts, diss_acts)
 2:     for layer in model.layers do
 3:         layer.prob_elems ← 0
 4:         for element in layer.elements do
 5:             sim_diff, diss_diff ← [], []
 6:             // Gather element-wise activations.
 7:             for sim in sim_acts do
 8:                 sim_diff ← sim_diff ∪ (sim.tgt[element] - sim.src[element])
 9:             for diss in diss_acts do
10:                 diss_diff ← diss_diff ∪ (diss.tgt[element] - diss.src[element])
11:             kw_out ← KWTest(sim_diff, diss_diff)
12:             // If the null hypothesis of
13:             // same distribution is rejected,
14:             // then the element counts as problematic.
15:             if kw_out.p_value < 0.05 then
16:                 layer.prob_elems++
17:     // Sort layers in descending order of problematic elements.
18:     return sort(model.layers,
                (l1, l2) ⇒ l1.prob_elems − l2.prob_elems)
```

Weights and Biases (WB)

We extract the weight and bias tensors for each of the convolutional layers. Starting from the most suspicious layer in the ranked list, we compare the corresponding *Source* and *Target* tensors element-wise for each of the layers. We flatten the tensors to facilitate comparison. For correct model conversions, we expect the weights and biases for the *Source* and *Target* model layers to match exactly. Values for model weights and biases are only affected during model training or optimization, neither of which is part of model conversion. In addition to convolutions, we also perform comparison of weights and biases in neighboring layers, e.g., batch normalization nodes preceding and/or succeeding the convolutional nodes). Consequently, differences identified during the comparison of weights and bias tensors for layers across *Source* and *Target* point to potential problems in those layers. We output all the layers that have differences identified in weights and bias tensors between *Source* and *Target* and pass it to the repair component.

Layer Hyperparameters (LH)

Incorrectly converted hyperparameters are another potential source of error. For example, with a convolutional layer, we expect that the padding, strides, dilation, and other configurations to remain unchanged during model conversion. Starting from the most suspicious layer in the ranked list, we compare layer hyperparameters and attributes – (`pads`, `strides`, `kernel_shape`, `dilations`, `epsilon`, `min`, `max`, `axis`) between *Source* and *Target*. If we find any differences, we mark them as potential faults. In particular, we observe 3 types of differences: (1) a hyperparameter is present in *Source* but not in *Target*; (2) a hyperparameter is present in *Target* but not in *Source* (e.g., added by the converter tool); and (3) a hyperparameter is present both in *Source* and *Target*but their values are different.

Computation Graph (CG)

In our experiments, we found that output label discrepancies can sometimes arise from differences in model graph between *Source* and *Target* models. We examine differences in graph nodes. To make the analysis tractable, we start from the most suspicious layer (based on the ranking from layer activation analysis) in *Target* and extract the subgraph between this layer and its dominator in the *Target* model graph.

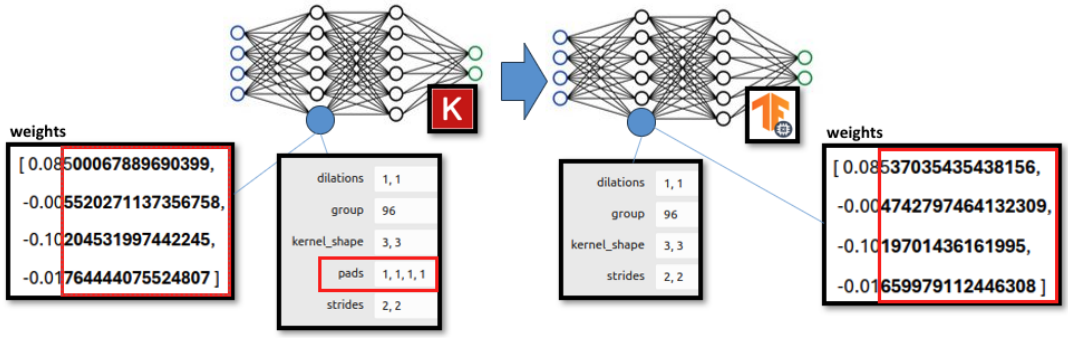


Figure 4.2: Indicative example of differences in layer weights and hyperparameters, introduced in the model conversion process.

We do the same with the *Source* graph starting from the corresponding *Source* layer node and its corresponding dominator. We perform a depth first traversal of the two subgraphs and compare their nodes. Any differences are marked as a potential fault. We repeat this for any other suspicious layers in the ranked list.

While `FetaFix` does not currently support complex network metrics, these metrics could be leveraged to detect graph differences more effectively. For example, node degree and clustering coefficient can help identify significant structural differences within the model graph. Although this is beyond the scope of the present work, we consider it a promising direction for future extension.

4.2.2 Fault Repair

We implemented a number of strategies to repair the potential faults identified by the fault localization component. We discuss the strategies to fix each of the issues identified in fault localization below. As seen in Figure 4.1, for layer-based repairs to CG, LH, and WB, the fault repair component will start from the most suspicious layer in the layer ranking provided by the layer activation analysis (within fault localization) and apply fixes sequentially and iteratively to layers in that ranking. Each of the fixes to *Target* produces an updated version. The updated *Target* will only be accepted if the output label discrepancy with respect to *Source* reduces after the repair (compared to previous target version), and this is then used as the current *Target* for the next repair strategy or suspicious layer in the list. On the other hand, for PP, ID, TSS, the repair actions are layer-independent, and do not need to consider the layer ranking. The repair strategies are applied sequentially with PP applied first, followed by ID and then TSS. As with layer-based strategy, the fix is only accepted. In particular, if the output label accuracy improves between *Source* and *Target*, the "most dissimilar"

image (as defined in Section 4.2.3) is initially selected and used after each repair attempt, with the goal of maximizing output label prediction similarity between *Source* and *Target* as a local maximum. Once this local maximum is reached, the repaired model is evaluated on the entire test dataset to maximize output label prediction similarity globally between *Source* and *Target*. Finally, both input-based and layer-based strategies are applied to every image in the dataset that had different labels between *Source* and *Target* until they are all fixed or the algorithm times out.

Similarly to fault localization, we combine one-off repairs with greedy Search-Based Fault Localization (SBFL) [239, 240] in order to repair the *Target*model. Using the layer ranking and fault localization information, `FetaFix` performs one-off repairs for the input-based strategies, and SBFL for the layer-based strategies, using one sample input as the sample case that repairs will be tested against quickly. In case a repair is proven to be effective to this sample (improving its accuracy), then the potential fix will be tested against the whole dataset, and if found effective, it will be integrated to a new baseline *Target*. The process is repeated until *Target* is fully repaired, or `FetaFix` cannot apply any additional repairs.

Input-based Repair Strategies

Preprocessing Differences (PP) Repair We experiment running the *Target* model with both the *Target* and *Source* preprocessing settings to check if the label discrepancies are resolved.

Input Dimensions (ID) Repair Following fault localization, if a difference is detected between *Source* and *Target* input dimensions, `FetaFix` automatically fixes *Target* to match with the input dimensions from *Source*. We observed four erroneous model conversions in our experiment with incorrect input dimensions, with two of them requiring further actions in model nodes for tensor shape and structure (discussed below).

Tensor Shape and Structure (TSS) Repair Following fault localization, as described in Section 4.2.1, if the input is different between *Target* and *Source* in a problematic conversion, and input is succeeded by a `Transpose` node, then `FetaFix` attempts to apply a fix by removing that node, in addition to adjusting the *Target* input to match *Source*. Furthermore, if `FetaFix` detects any nodes that are related to input processing and normalization (e.g., InceptionV3 architecture contains `Gather`, `Unsqueeze`, `Add`, and `Mul` nodes before the added `Transpose`), then `FetaFix` adjusts

the attributes of these nodes in order to be compatible to the *Source* input dimension. In cases where `FetaFix` detects the presence of `Flatten` or `Reshape` nodes that alter the shape of tensors, then it attempts to repair this by inserting a `Transpose` node right before `Flatten` or `Reshape` to match tensor shape with *Source*.

Layer-based Repair Strategies

Weights and Biases (WB) Repair If a mismatch is found between any weight or bias tensor across model layers by the fault localization component, then `FetaFix` replaces the values from the corresponding tensor in *Source* to *Target*. As mentioned earlier, `FetaFix` performs the repair starting from the most suspicious layer in the ranked list.

Layer Hyperparameters (LH) Repair Upon marking a layer hyperparameter case as potentially problematic, as described in Section 4.2.1, `FetaFix` attempts to fix *Target* by applying the corresponding setting of *Source*. In particular, if (1) a node is present in *Source* but is missing from the computation graph of *Target*, `FetaFix` will update the corresponding node including the *Source* hyperparameter to *Target*. If (2) a hyperparameter is added in *Target* but is missing from *Source*, `FetaFix` will remove it from the target node. And finally, if (3) there is a value mismatch between *Source* and *Target* for the hyperparameter, `FetaFix` will update *Target* in accordance to the source value.

Computation Graph (CG) Repair In accordance to Section 4.2.1, if a difference in subgraphs is detected between *Source* and *Target*, `FetaFix` attempts to repair it by replacing the *Target* subgraph nodes with *Source* subgraph nodes, preserving all the node properties and ensuring all weights, biases and hyperparameters are correctly updated in the *Target* graph.

Regarding the order in which our repair strategies are applied, we first applied the ones that were input-based, namely PP, ID, and TSS. We do PP first as changes to preprocessing may fix errors observed downstream. We then examine ID, with TSS being an additional restriction to ID mismatches. We then apply repairs that are "layer-based" – WB, LH, and CG. Among the ones with layer-based effects, we first perform repairs related to model weights, biases and hyperparameters, and then those related to computation graph modifications. Note that some graph transformations may also be related to hyperparameter changes, e.g., removing a *Target* `Pad` node to adapt to *Source* structure requires the addition of a `padding` hyperparameter to the next convolutional layer.

4.2.3 Algorithm Overview

Algorithm 2 presents our repair approach in FetaFix. We describe in detail each one of the important parts in the following sections.

Setup

As presented in lines 2-8, FetaFix performs inference of *Source* model against the test dataset (imgs), while also building the *Source* model in TVM and setting its configuration. In line 5 the TVM model for *Source*is built, while in lines 7 and 8, FetaFix loads the configuration and initializes same_diss=0, which is the counter of times that the same dissimilarity percentage is allowed in the system, incrementing and resetting accordingly across repair iterations. Once this variable reaches cfg.diss_no times of iterations that the dissimilarity percentage remains unchanged (i.e., the repair is ineffective), then the process terminates. In addition, FetaFix initializes the dissimilarity percentage (diss_perc) to 100. Finally, the system loads the maximum execution time from configuration in variable t. Other initial setup variables include cfg.analysis_iter_no, that manages the number of iterations of the layer activation analysis for suspicious ranking, total_diss_imgs which holds a set of "mostly dissimilar" images selected for the repair process (a selection process explained in Section 4.2.3), as well as Nsim and Ndiss, that indicate the number of images with similar and dissimilar labels that is being considered. Next (line 9), FetaFix performs inference with *Target* over imgs, and compares inference labels between *Source* and *Target* to obtain the dissimilarity percentage diss_perc (line 12).

Selection of Images Under Test

In lines 15-20, FetaFix selects the "most dissimilar image", by ranking the dataset images that were executed across *Source*and *Target*and presented different results across the two variants. Specifically, FetaFixranks the dataset image inputs by comparing the label ranking for each image between *Source* and *Target*and measuring the extent of difference. Label rankings are compared using Kendall's Tau (KT), by considering a set of top-K predictions for each input across *Source* and *Target*, with default $K = 5$. This way, a correspondence score can be calculated across the two sets of predictions between *Source* and *Target*, for each image, determining the extent of the dissimilarity observed between model variants. KT has a value range of -1 to 1 with 1 indicating perfect match between *Source* and *Target* label rankings for a single image). This way, the most dissimilar image (the one with the lowest KT score) is

Algorithm 2 Repair Approach in `FetaFix`

```
1: procedure repair(src_onnx, tgt_onnx, imgs, cfg)
2:     strats ← ["params", "hyperparams", "graph"]
3:     src_res ← inference(src_onnx, imgs, cfg)
4:     all_eval ← None
5:     src_tvm ← build_tvm(src_onnx, cfg)
6:     total_diss_imgs ← []
7:     same_diss, diss_perc ← 0, 100
8:     t, dn ← cfg.time, cfg.diss_no
9:     while diss_perc ≠ 0 or same_diss < dn or !t.elapsed() do
10:        tgt_res ← inference(tgt_onnx, imgs, cfg)
11:        all_eval ← eval(src_res, tgt_res)
12:        diss_perc ← all_eval["diss_perc"]
13:        same_diss ← times_appear(diss_perc)
14:        // Most Dissimilar Image Selection.
15:        diss_img ← sel(all_eval["diss"], sort="KT", excl=total_diss_imgs)[0]
16:        total_diss_imgs ← total_diss_imgs ∪ diss_img
17:        tgt_tvm ← build_tvm(tgt_onnx, cfg)
18:        // Input-Based Repair Strategies
19:        md ← Modifier(src_onnx, tgt_onnx)
20:        tgt_upd ← md.apply_once("preproc", cfg)
21:        if is_fixed(tgt_upd, diss_img) then
22:            tgt_onnx ← tgt_upd
23:            continue
24:        if src_onnx.input ≠ tgt_onnx.input then
25:            tgt_upd ← md.apply_once("repair_input", cfg)
26:            if is_fixed(tgt_upd, diss_img) then
27:                tgt_onnx ← tgt_upd
28:                continue
29:        tgt_upd ← md.apply_once("fix_transpose", cfg)
30:        // Suspicious Layer Ranking.
31:        dbg ← tvm_debug(src_tvm, tgt_tvm)
32:        prs_a ← compare_params(dbg.src_params, dbg.tgt_params)
33:        lrs_a ← []
34:        for i in cfg.analysis_iter_no do
35:            sim_imgs ← rand_sel(Nsim, all_eval["sim"])
36:            diss_imgs ← rand_sel(Nsim, all_eval["diss"])
37:            run = infer_tvm(src_tvm, tgt_tvm, sim_imgs, diss_imgs)
38:            lrs_a ← lrs_a ∪ layer_analysis(src_tvm, run.sim_acts, run.diss_acts)
39:        layer_order ← rank_order(prs_a, lrs_a)
40:        // Layer-Based Strategies Repair.
41:        local_best ← tgt_onnx
42:        for strat in strats do
43:            for layer in layer_order do
44:                md.load(local_best)
45:                tgt_upd ← md.apply(strat, layer)
46:                if is_KT_improved(tgt_upd, diss_img) then
47:                    local_best ← tgt_upd
48:                    if is_fixed(tgt_upd, diss_img) then
49:                        tgt_onnx ← tgt_upd
50:                        break
51:    save_metadata(tgt_onnx, all_eval)
```

selected in the diss_img variable (following ascending order ranking (line 15). Once an image is considered for testing, FetaFixmarks it as used and does not apply it again for testing. It is worth noting that this comparison approach is different from the one we followed in DeltaNN, as there, it was sufficient to only focus on the top-1 class prediction output to determine dissimilarity percentages across model variants. However, in FetaFix, we follow a step-by-step, iterative repair process, where the extent of which a prediction is "improved" is taken into consideration, so that we also consider model repairs that partially improve the *Target*model correctness.

Input-Based Repair Strategies Application

As a next step, FetaFix performs repair for input-based fault types that were detected. Specifically, FetaFix performs inference with different preprocessing settings (lines 19-23), while it also attempts to repair issues with input dimensions (lines 24-28) and tensor shape and structure (line 29), generating a new *Target* variant, tgt_upd with each fix. FetaFix then performs inference with the dissimilar image dataset, diss_imgs, then calculates and compares its KT using the top-K predictions of the new *Target*variant and the original *Target*, or the last fixed version (containing fixes that improve model correctness in comparison to the initial *Target*). If there is an improvement in the KT score with tgt_upd from the last fixed version, then we set tgt_upd to be the latest fixed version.

Suspicious Layer Ranking

Lines 31-39 present the part of the analysis process utilized for layer-based fault localization, containing parameter and layer activation analysis, as described in Section 4.2.1. First, using TVM builds of models, src_tvm and tgt_tvm, the system then performs parameter extraction for model layers, using TVM Debugger and adding it to the object (dbg). Then, FetaFixcompares parameters statically across TVM model variants once by comparing the parameter values metadata extracted by the TVM Debugger and storing the differences in prs_a variable (line 31). As a next step, FetaFixperforms layer activation analysis using the layer_analysis function presented in Figure 1. This is performed cfg.analysis_iter_no times, defined in configuration (contained in the cfg object). For each iteration, FetaFix randomly selects a set of Nsim images from the set of inputs that resulted in similar predictions across *Source*and *Target*, and another set of Ndiss random images from the set of inputs that resulted in dissimilar predictions. Following this step, it will invoke

the `layer_analysis` function (presented in Algorithm 1), and generate a layer activations rank, which will be appended to the `lrs_a` variable. The total analysis output is stored (variables `prs_a` for parameters and `lrs_a` for all activation analysis iterations) and combined using *Average Rank Order* in order to determine the ranking of layers for fault repair (line 39).

Layer-Based Strategies Repair

Lines 41-50 depict the layer-based application of strategies. More specifically, the tool iterates through the strategies, defined in Section 4.2.2. For each strategy `strat`, `FetaFix` iterates through the suspicious layers of *Target* based on the ranking and applies a repair attempt. Each repair, will result in a new *Target*version, `tgt_upd`.

Following each repair with `tgt_upd` generation, `FetaFix` checks the KT score for `diss_img` and accepts `tgt_upd` to be the latest fixed version if there is an improvement in KT score. Otherwise, `tgt_upd` gets discarded. However, if the score is positive and close to 1 (`KT>=0.99`), then the system stops the iterative repair process as the discrepancy for that image is considered fixed. The algorithm then goes to the start of the loop in line 9, repeating the process for other dissimilar images within the `diss_imgs` dataset until the termination condition is met.

Termination Condition

As presented in line 9 of Algorithm 2, `FetaFix` repeats the process iteratively until at least one of the following criteria is met:

- The *Source* and repaired *Target* dissimilarity percentage for the given dataset reaches zero,
- the dissimilarity percentage remains the same over a predetermined number of attempts, `cfg.diss_no`, that is defined in the configuration, or
- a predetermined amount of time (in seconds) elapses, set as (`cfg.time`) during setup.

The system then concludes the process by generating a repaired model with information on all the fixes applied (line 51) and the inference outputs for *Source* and the repaired *Target*.

Implementation Details

We implemented `FetaFix` in Python. We used Apache TVM Debugger for layer activations analysis. The debugger allows the extraction of model graph structure and parameters in metadata files, and a log of the layer activations throughout the model structure. In order to implement the repair strategies, we used the ONNX API to modify *Target*.

4.3 Experiments

We consider three widely used image recognition models of various sizes: MobileNetV2 [71], ResNet101V2 [66, 67], and InceptionV3 [72]. We use models pre-trained on ImageNet [68] using native model definitions and pre-trained parameters/weights sourced from four different DL frameworks' repositories: *Keras* [47], *PyTorch* [46], *TensorFlow(TF)* [51], and *TFLite* [51]. For our experiments, we used two datasets: the ILSVRC2017 [69] object detection test dataset consisting of $5,500$ images, and the ImageNetV2 dataset [283] consisting of $10,000$ images.

We used the ILSVRC2017 dataset for detecting and repairing errors in model conversion. We then assessed the correctness of the repaired models on ImageNetV2, utilizing it as an independent dataset. We used the problematic conversion cases identified in Chapter 3 as our candidates for fault localization and repair. Both datasets contain real, non-synthetic samples and they are widely utilized by developers and researchers, while remaining tractable in size. We focused on cases where the conversion process introduced label discrepancies between *Source* and *Target*, and there were 15 such erroneous conversions, as seen in Figure 3.4.

Overall, we selected models in the domain of computer vision for the following reasons: (1) To test our repair approach against the conversion faults we observed using `DeltaNN` [70]; and (2) since our experiments involved 15 model conversion cases with numerous iterations and analysis within each, we chose image recognition models as they are more tractable and need less resources than other DL architectures, like transformers, for our experiments. However, our approach can generalize easily to any DL framework or conversion tool potentially even addressing the issues originally sourced from Q&A websites and forums, which were used to establish the fault localization and repair strategies of `FetaFix`. It can be applied to any DL model architecture, including non-CNN models, only requiring the user to specify the core computational layers to be used in the comparison.

We evaluate the following research questions in our experiments:

RQ1: How effective is FetaFix at localizing faults in the 15 erroneous model conversions?

RQ2: How effective is FetaFix at repairing the faults in the 15 erroneous model conversions? We evaluate which of the repair strategies are used most frequently in model fixes.

To check the robustness of FetaFix with respect to devices, we ran our experiments on two devices of varying capabilities, an Intel-based server featuring a high-end Nvidia Tesla K40c (GK11BGL) GPU, and a Laptop featuring a low-end Intel(R) GEN9 HD Graphics NEO. Our experiment results remained the same on both devices.

4.3.1 Fault Analysis Setup

Within layer activation analysis, discussed in fault localization for layer-based types, we determine the expected distribution of differences in activations per layer between *Source* and *Target* for images where the output label is the same using a random sample of up to 100 images that produced similar results. We then compare the activations for dissimilar images against this expected distribution to come up with a ranking of suspicious layers. We repeat this process 3 times with three different random samples of similar images.

4.3.2 Fault Repair Setup

Once we extracted the order of layers from the fault analysis, we utilized the fault repair component of FetaFix, applying the strategies in a greedy approach, as described in Section 4.2.2. We set a time limit of 2 hours for each conversion case. We also set a limit of three unsuccessful repair iterations, when repairs did not reduce output label discrepancy, as a stopping condition. This is to help make the experiments tractable. These settings can be customized as needed in our tool.

4.4 Results

Overall, FetaFix was able to localize 755 differences in the structure and properties between *Source* and *Target* variants across all 15 erroneous conversion cases. We identify these differences as potential faults - given that (1) they might not affect model correctness (false positives) and their impact needs further examination, and (2) many of those differences might have common roots of cause in properties, structure, as well as the converter tools, affecting *Target* models during the conversion process. FetaFix attempts to distinguish which of those differences had an actual impact to the model correctness and only repair those, while discarding the rest of them. Furthermore, an overview of the potential faults distribution can be found in Table 4.2. From these 755 cases FetaFix identified as potential bugs, it correctly distinguished as actually problematic and repaired 462 of them — completely repairing 12 out of 15 problematic model conversions, while significantly improving the accuracy of 2 more, with the most distinctive case being improved from 48.9% output label discrepancy to 0.33%. Our system was unable to reduce the percentage further, as none of our strategies affected the model beyond this point, and unfortunately could not identify the causes of the remaining output label discrepancy, thus reaching its limitations.

4.4.1 Fault Localization

The faults localized were distributed across all six types within input-based and layer-based fault categories. Of the 755 faults localized, 462 of them (61.1% of total cases) resulted in reducing output label discrepancy between *Source* and *Target* when repaired. The remaining localized faults did not result in any improvement after repair and are treated as false positives detected by the fault localization component. We demonstrate the distribution of faults detected and fixed across the different types in Table 4.2 and discuss each type in the context of fault localization next.

Overall, the faults we detected are new, as we detected no reported issues with relation to them in the respective converter repositories. We aim reporting the issues to the respective developers of the conversion tools.

Preprocessing (PP): We detected 9 cases in which the choice of preprocessing between *Source* and *Target* configurations was a decisive factor for the model performance, while 5 cases presented the same results across *Source* and *Target* configurations. All models presented severe accuracy degradation in output correctness if no preprocessing option was selected.

Table 4.2: Number of faults localized versus repaired (#Localisations/#Repairs) per model conversion case.

		#Localisations/#Repairs					
		Input-Based			Layer-Based		
#	**MobileNetV2**	PP	ID	TSS	WB	LH	CG
1	Keras-to-Torch	**1/1**	0/0	0/0	0/0	**6/0**	0/0
2	Keras-to-TFLite	**1/1**	0/0	0/0	**96/96**	0/0	0/0
3	Torch-to-TF	**1/1**	0/0	0/0	0/0	**6/0**	0/0
4	Torch-to-Keras	**1/1**	**1/1**	**1/1**	0/0	**39/0**	0/0
5	TF-to-TFLite	0/0	0/0	0/0	**68/66**	0/0	0/0
6	TF-to-Keras	**1/1**	0/0	0/0	0/0	**40/0**	**4/0**
	ResNet101						
7	Keras-to-Torch	**1/1**	0/0	0/0	0/0	**72/0**	**2/0**
8	Keras-to-TFLite	0/0	0/0	0/0	**180/174**	**33/0**	**37/0**
9	Torch-to-TFLite	**1/1**	**1/1**	0/0	0/0	0/0	0/0
10	Torch-to-TF	**1/1**	0/0	0/0	0/0	0/0	0/0
11	TFLite-to-Keras	0/0	0/0	0/0	0/0	**4/4**	**3/3**
12	TF-to-TFLite	0/0	0/0	0/0	0/0	0/0	0/0
13	TF-to-Keras	0/0	**1/1**	**1/1**	0/0	**1/1**	**1/1**
	InceptionV3						
14	Torch-to-Keras	**1/1**	**1/1**	**1/1**	0/0	**54/7**	0/0
15	TF-to-TFLite	0/0	0/0	0/0	**94/94**	0/0	0/0
	Total	**9/9**	**4/4**	**3/3**	**438/430**	**254/12**	**47/4**

Input Dimensions (ID): `FetaFix` identified four model cases where the input dimensions were different between *Source* and *Target*. We identified that input dimension errors had two primary causes: (1) erroneous configuration from the user, which neither the model converters nor the model itself raised as warnings or errors; (2) implicit changes in model input due to the converter implementation.

We observed two cases, 9 and 13 in Table 4.2, where ID fault is a result of the first cause. Regarding the second cause, we detected two cases where the converter changed the model input. In addition to ID faults, we also detected differences in tensor shape and structure that we discuss as part of the next category.

Tensor Shape and Structure (TSS): `FetaFix` detected three cases with potential faults in the TSS category. The model converter introduced a `Transpose` node in these two cases to address the difference in ID so that subsequent nodes use inputs with the correct dimension. However, in these two cases getting to the correct input

dimension with the Transpose node was not adequate. This occurred due to similar sized dimensions within tensors, like the third and fourth dimension within an example tensor with shape (1, 3, 224, 224), where it is possible to incorrectly transpose dimensions of the same size that result in a different structure from what is intended.

In addition, in some model architectures (e.g., InceptionV3) the Transpose node is placed after a series of nodes that normalize data based on a specific input dimension. For instance, in InceptionV3, the Transpose node is placed after a series of Gather, Unsqueeze, Mul, and Add nodes. FetaFix detects these differences across model graph and reports them during fault localization.

Weights and Biases (WB): We observed 438 cases of differences in tensor values between weights and biases across *Source* and *Target*, with ResNet101 converted from Keras to TFLite containing 177 of them.

Layer Hyperparameters (LH): FetaFix detected 254 potential faults across 12 model conversion cases, with most of them concerning padding hyperparameters. FetaFix also identified some cases where a potential hyperparameter fault was associated with a potential computation graph (CG) fault - for instance, specific convolutional layers presented potential faults related with padding hyperparameters and Pad nodes that directly preceded these convolutional layers.

Computation Graph (CG): Our system identified 47 cases across 4 conversion cases in which parts of the model computation graph presented discrepancies across *Source* and *Target* variants. All the cases observed were associated primarily with three operations: (1) setting a padding around a tensor, by utilizing Pad nodes; (2) performing batch normalization to a tensor by applying an element-by-element constant addition, and scaling using a BatchNorm node; and (3) redundant output nodes erroneously placed by the converter. From those 47 cases, FetaFix successfully identified 4 of those cases as actual bugs and repaired them, playing a crucial role for the repair of cases $\#11$ and 13. For cases initially localized as potential faults but ultimately not identified as actual issues, we concluded that they involved differences in the computational graph between *Source* and *Target*. However, these differences were semantically equivalent, meaning that the variants performed the same operations in practice.

4.4.2 Fault Repair

From the 755 potential faults localized and reported in Table 4.3, FetaFix successfully repaired 462 of them across the 15 erroneous mode conversion scenarios. In 12 out of the 15 cases, FetaFix was able to repair the model completely, starting from up to 100% initial output label discrepancy. In two other cases, FetaFix was effective reducing the output label difference between *Source* and *Target* to less than 1.5%. The only case that FetaFix was unable to repair was because the *Source* and *Target* models could not compile in the same opset, therefore the graph representation of the model was different, deeming it unable to detect potential model discrepancies. By performing manual inspection across both *Source* and *Target*, we observed that *Target* was representing the weights and biases of each layer in separate nodes, while on *Source* weights and biases were associated as layer inputs. FetaFix is implemented to match weights and biases represented as inputs to each layer, and therefore was unable to repair this scenario. However, since the model opsets were different, FetaFix was effectively comparing different representations, essentially in a manner of comparing "apples to oranges", with the model graph in the one variant being represented using multiple nodes in the other. As a result, it was unable to properly match layers across variants. As future work, this limitation could be addressed by enhancing the layer matching capabilities of FetaFix to support one-to-many node mappings (i.e., where a single node in one model corresponds to multiple nodes in the other, across *Source* and *Target*). We discuss fault repair with respect to each of the fault types below.

PP: As shown in Table 4.3, FetaFix was able to repair cases $\#1$, 3, 6, 7, 9 and 10, by determining and setting the correct preprocessing configuration. Given that the preprocessing is not a standard procedure across models, we identified that for these cases using the *Source* preprocessing instead of the one officially provided for the *Target* library fixed the result completely. Cases $\#2$, 4, 9, and 14 were also affected partially by preprocessing selection, however additional steps were needed in order to repair them. The additional steps are explained in more detail in the categories below.
ID: Among the 4 faults localized as ID, FetaFix repaired one case completely by solely adjusting the input dimensions of *Target* to match *Source*. For the remaining three cases, additional actions had to be performed, involving repair tensor shape and structure, as well as computation graph repair, as described in the next categories.

TSS: FetaFix was able to repair three cases related to problematic tensor shape and structure. In particular, cases 4, 13, and 14 were related to problematic layers right after input. All three cases had different input dimensions between *Source* and *Target*, and the conversion tool had inserted additional Transpose nodes in order to adjust the input tensor shape so that it would be aligned with subsequent nodes. However, the transformation was done improperly, leading to severe accuracy degradation of up to 68.21%. We identified all cases to be related to the onnxmltools [284] converter (part of the native ONNX codebase), when converting Keras to ONNX. FetaFix automatically identified the presence of a redundant Transpose node following the input node, and repaired it by (1) adjusting the *Target* model input dimensions so that they correspond to those of *Source*, (2) altering its tensor transposition order (permutation) property to align with the new input dimensions, and (3) updating the properties of nodes connected to Transpose to ensure proper tensor handling, reflecting the change in Transpose. This modification in particular was vital to case $\#14$ where FetaFix detected additional nodes (Gather, Unsqueeze, Add and Mul) preceding Transpose that apply input normalization. As part of the repair process, FetaFix updated the axes properties of Gather and Unsqueeze nodes to align with the updated input dimension and the Transpose permutation in *Target*.

WB: Cases $\#2$, 5, 8, and 15 were related to parameter precision discrepancies between *Source* and *Target*. Although the discrepancies were negligible, the model behavior deviated across the cases from 3.12-9.7%. FetaFix performed multiple iterations and managed to completely fix cases $\#5$ and 15 while also significantly improving cases 2 and 8 by replacing the weights of *Target* with those of *Source* where differences were detected. We identified all cases to be related to the behavior and the setup of TFLiteConverter. The discrepancy after each repair step is depicted in Figure 4.3. At each repair step, FetaFix adjusts the weights & biases of a suspicious layer in *Target* with those from *Source*, in the order determined by the Layer Activation Analysis (Section 4.2.1), and evaluates the effects of the change by comparing the predictions of *Source* and *Target*.

LH: From the 254 potential faults FetaFix localized, only 12 of them were actively affecting the model. These hyperparameters were associated with the pads hyperparameter of Conv and MaxPool nodes (cases $\#11$ & 13), as well as with the axes of Gather and Unsqueeze nodes responsible for input processing (case $\#14$). Repairing *Target* entailed adjusting their values with those from *Source*.

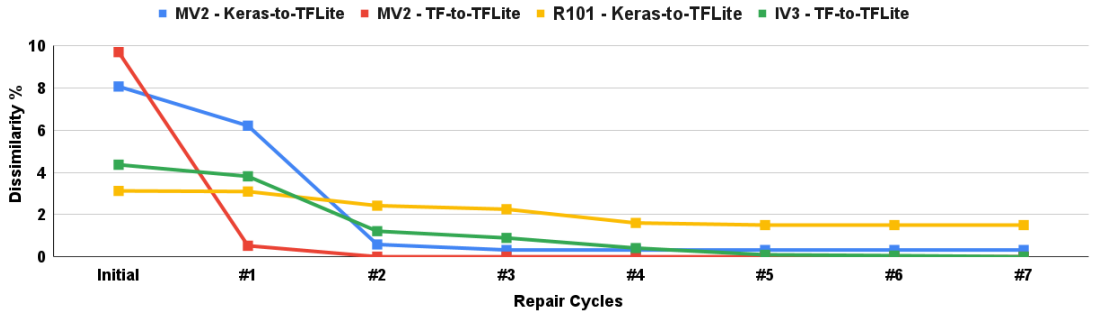


Figure 4.3: Model conversion cases that required an iterative weights and biases repair strategy with the percentage output label dissimilarity shown after each repair cycle (following preprocessing repair).

CG: `FetaFix` identified 47 cases of graph-related potential faults. Of these, 4 cases resulted in fault repairs that improved model accuracy (reduced output label discrepancy). The repairs played a crucial role in conversion cases $\#11$ and $\#13$ that initially presented discrepancies of 100% and 35%, respectively. Our tool identified that, for case $\#11$, the converter had erroneously introduced multiple output nodes. In addition, three problematic padding nodes were found to affect the behavior of the models. `FetaFix` identified such faults and completely repaired the models by (1) removing any additional output nodes added to the model by the converter, and (2) repairing problematic padding nodes.

For all the false positive cases observed in LH and CG, we performed manual inspection and observed that the converters performed semantically equivalent computation graph conversions. For instance, in the erroneous conversion presenting the largest number of CG fault localizations – ResNet101, `Keras` to `TFLite` (case $\#8$) – *Source* contained `Pad` nodes which were absent in *Target*. However, *Target* contained a `pads` hyperparameter in the next convolutional layer that had the same effect. Also, `BatchNorm` nodes of *Source* were replaced by the converter with `Mul` and `Add` nodes in *Target* effectively applying the normalization in a two-step semantically equivalent manner. `FetaFix` attempted to repair these scenarios by replacing them with the corresponding *Source* subgraph but these fixes had no effect on the number of output label discrepancies encountered.

In terms of the number of iterations needed during the repair process, input-based strategies are only applied once. Layer-based strategies typically require multiple iterations. Four erroneous model conversions that required multiple repair iterations related to WB strategy repair are presented in Figure 4.3, ranging from 2 to 7 repair cycles, using the ILSVRC2017 dataset.

4.4.3 Fault Localization/Repair Effectiveness

We consider effectiveness as the number of localized faults that could be repaired to improve *Target* prediction accuracy towards matching this of *Source*. We go through each fault category to determine the effectiveness of the fault localization component and are automatically discarded from the repair process, as seen in the `Total` row in Table 4.2. We find that `FetaFix` had an 100% efficacy rate (all potential fault occurrences detected could be repaired) for preprocessing (PP: $9/9$), input dimensions (ID: $4/4$) and tensor structure (TSS: $2/2$), as well as 98.1% efficacy for weights and biases (WB: $430/438$). The weights and biases category had the most faults detected and repaired. `FetaFix` had much less success on cases related to the computation graph (CG), with 4 of the 47 potential faults detected being repaired (8.5%) while other cases when repaired did not result in reduction of output label discrepancy. The CG fault repairs, although few in number, were instrumental in completely fixing conversion cases $\#11$ and 13. The effectiveness for LH category was low (4%), with only 12 out of the 254 identified potential faults worth repairing. Following manual inspection, we infer that this happens because: (1) graph transformations across *Source* and *Target* may result in syntactic difference but be semantically equivalent. For instance, a `BatchNorm` node is transformed in `Mul` and `Add` nodes. Although this is localized as a potential fault by `FetaFix` the repair employed by replacing `Mul` and `Add` nodes with `BatchNorm` will not reduce the output label discrepancy and is therefore rejected; (2) the LH difference detected is already mitigated by the converter by introducing additional nodes in the *Target*. For example, a `pads` (padding) hyperparameter in a `Conv` node on *Source* is replaced by a dominating `Pad` to the correspondent `Conv` node in target, leading to the same effect.

In terms of the fault analysis for layer-based types, we set `FetaFix` to perform fault localization with and without fault analysis, and we compared the outcomes. We concluded that `FetaFix` was able to repair the models in a similar manner with the results presented without the fault analysis, however enabling it applied fewer fault repairs, more effectively avoiding false positives (with up to 3.6% reduction). We also found the layer ranking assisted towards applying smaller repair steps, with the dissimilarity percentage decreasing more gradually with each repair cycle, making it easy to monitor and control the fault process.

4.4.4 Repair Validity

In order to verify our results, we used the ImageNetV2 [283] dataset consisting of 10K images which was built with the purpose of providing test data for ImageNet, as an independent validation source. We checked the discrepancies of the repaired *Target* models against *Source*, and verified that (1) all cases that were fully repaired using ILSVRC2017 also presented no discrepancies with ImageNetV2, and (2) partially repaired cases $\#2$ and 8 presented 0.28% and 0.66% discrepancies, lower than those observed in ILSVRC2017. The full comparison can be found in Table 4.3. In addition, we used ImageNetV2 to verify that repaired models were giving consistent results against the dataset ground truths. Finally, we tested all 10 cases shown in Figure 3.4 presenting no discrepancies with ImageNetV2, to verify that no additional errors were detected. In detail, we verified in our experiments that the results of each repair cycle do not misclassify previously correct results, by testing all repaired models on ImageNetV2, checking the classification in two ways - against ground truth for the dataset inputs, and by comparing the results between source and target models. We did not observe any misclassification cases of correct *Target* model predictions prior to the repair process across the entirety of our experiment set.

4.4.5 Root Causes of Faults

We consider that many of the errors detected might have common root causes as part of the conversion process. Upon further manual examination, we observed that (1) all faults related with preprocessing could be repaired when the proper setting was selected, (2) errors related to Weights & Biases (WB) were related to an implicit action caused by `tf2onnx` converter, and (3) issues related to Input Dimensions (ID), Tensor Shape & Structure (TSS), Layer Hyperparameters (LH) and Computational Graph (CG) were attributed to problematic layer handling by converters (e.g., `onnx2tflite`). We reported such issues to the development teams of each converter tool.

4.4.6 Results Generalizability

We expect our approach to be applicable to any DL frameworks containing core computational layers and their conversion tools - including non-CNN models, requiring from the user to specify the core computational layers to be used in the comparison. In addition, the user must define a comparator for the evaluation of the model type under test (e.g., BLEU [285] for text generation models). Providing proof of FetaFix effectiveness in this scenario, is subject to future work. In terms of our experiments, we chose image recognition models in our experiments as they are more tractable and need less resources than other DL architectures like transformers, as our experiments involved 15 model conversion cases with numerous iterations and analysis within each. All the models selected are widely used, either as-is or as the base for extension of models deployed on more complex tasks (e.g., ResNet101 extended for object detection). Our current implementation is bassed on ONNX, as it is a widely-used format, providing a comprehensive API for model definition and modification. However, our methodology can be easily extended to other representations.

In terms of the fault types supported by FetaFix, they are based on a survey of issues encountered in discussion forums and empirical investigations (Table 4.1). There may be other fault types not currently considered by FetaFix. We are aware of this and we expect that FetaFix will keep evolving to handle emerging issues and other fault categories in model conversions between DL frameworks.

4.5 Discussion

Based on our results, we observed that FetaFix is highly effective for the automatic fault localization and repair of models converted across deep learning (DL) frameworks. However, despite its effectiveness, the generalizability and long-term value of FetaFixmight be questioned, under the argument that the engineering effort required to maintain and develop FetaFix could instead be invested in building more robust and reliable converters. We argue, however, that maintaining FetaFix is more efficient, as FetaFix is both DL framework- and converter-agnostic. FetaFix is not tightly coupled to any specific converter or DL framework. It can be used to repair models converted across any DL framework, provided that two primary requirements are met: (1) the converters must be able to extract the models in ONNX format, and (2) a comparison method must be available for the given model type. In addition,

Table 4.3: Experiments Overview: Discrepancies between *Source* and *Target*. before and after the repair process. for ILSVRC2017 and ImageNetV2 datasets.

		ILSVRC2017		ImageNetV2	
#	**MobileNetV2**	**Initial**	**Fixed**	**Initial**	**Fixed**
1	Keras-to-Torch	83.41%	**0%**	74.85%	**0%**
2	Keras-to-TFLite	49.34%	**0.33%**	31.87%	**0.28%**
3	Torch-to-TF	68.21%	**0%**	53.77%	**0%**
4	Torch-to-Keras	47.21%	**0%**	26.44%	**0%**
5	TF-to-TFLite	9.70%	**0%**	6.32%	**0%**
6	TF-to-Keras	48.90%	**0%**	31.56%	**0%**
	ResNet101				
7	Keras-to-Torch	36%	**0%**	89%	**0%**
8	Keras-to-TFLite	3.12%	**1.50%**	1.58%	**0.66%**
9	Torch-to-TFLite	72.30%	**0%**	53.41%	**0%**
10	Torch-to-TF	73.90%	**0%**	56.15%	**0%**
11	TFLite-to-Keras	100.00%	**0%**	100.00%	**0%**
12	TF-to-TFLite	2.70%	2.70%	1.32%	1.32%
13	TF-to-Keras	34.85%	**0%**	17.59%	**0%**
	InceptionV3				
14	Torch-to-Keras	32.38%	**0%**	15.89%	**0%**
15	TF-to-TFLite	4.36%	**0%**	1.53%	**0%**

`FetaFix` can be used to localize and repair faults in proprietary converters or even in models that have been manually converted across DL frameworks. Last but not least, `FetaFix` can be used not only to repair the converted models, but to identify and reproduce converter faults, so that they can be identified and repaired.

An additional concern related to `FetaFix` might involve its dependency to ONNX: if model repair relies on ONNX, this could introduce overhead and seemingly defeat the purpose of direct conversion across DL frameworks. We contend that this is not a limitation, but rather a practical design choice, as (1) ONNX is a widely adopted intermediate representation format for DNN models, and (2) most DL frameworks provide APIs to import and export models in ONNX format (e.g., `torch.onnx`).

Finally, another potential criticism is that FetaFix is limited only to classifier models. It is true that currently, FetaFix supports classification models, but it can be easily extended to support other types, such as object detection and text generation models, by defining and providing an Evaluator comparator class for the model type (e.g., measuring and comparing the F1 score [286] for object detection models or the BLEU metric [285] for text generation models).

4.6 Challenges and Limitations

Upon FetaFix development, we encountered a number of challenges and limitation, which we describe in this section.

Node Matching Across Source and Target In order to perform the fault localization and repair process, FetaFix had to correctly match the nodes across *Source* and *Target*. However, the node matching was not one-to-one, due to (1) changes in graph structure applied by converter tools, (2) modifications in shape and internal structure of model nodes and (3) alterations in layer names between the model variants. To mitigate this problem, we introduced a solid set of requirements that had to be fulfilled for two nodes to be marked as the same. In detail, we required that two model nodes were (1) of the same type, (2) of the same shape, and (3) they shared a number of parameters on which values could only deviate by a small margin of difference (absolute value of 0.2, obtained following step by step decrease to the point where the system failed to match the same nodes). For the moment, however, we match only core computational layers (e.g., convolutions), considering that their number is guaranteed to remain unaffected by the conversion process due to model design. We also limit the matching algorithm to nodes containing weights and/or biases, as the presence of such properties increased significantly the matching ability of FetaFix, to the extent of more than 95% across *Source* and *Target* variants. We aim to improve our system to this direction by performing matching considering the presence of the same hyperparameters across nodes between *Source* and *Target*.

ONNX Operation Limitations: For case $\#12$, we were unable to compile both *Source* and *Target* to the same opset, despite our efforts, due to ONNX operation incompatibilities. For that reason, we generated *Source* in opset=11 and *Target* in opset=13. As the two opsets had completely different graph representations, FetaFix was unable to match the layers and perform fault localization and repair.

As FetaFix relies on ONNX for its fault localization and repair capabilities, it has limitations as well, and therefore we were unable to perform comparison across the same opset variants. By manual observation, we concluded that this comparison limited our system from repairing the model, as the model structure across the two opsets was drastically different, to the extent that FetaFix was unable to match important nodes correctly (subject to the previous limitation aforementioned as well).

Computationally Intensive Fault Analysis: As the fault analysis we conducted performed per-element comparison for a number of inputs across each layer, it resulted in a vast amount of computations. This limited our tool from performing it to the full validation dataset, for reasons of tractability. For that matter, we selected a random sample of up to 100 images and we conducted our analysis on it, and repeated this process 3 times to solidify our results. We also accelerated the process by enabling parallel computation for the analysis, using Joblib python library [287]. One potential improvement for fault analysis is to adopt a search-based approach that begins with the model's input and output layers, and then recursively investigates adjacent layers that appear more "suspicious." This iterative narrowing allows the analysis to focus progressively on the most relevant parts of the network, rather than exhaustively analyzing all layers in the DNN.

4.7 Summary

We presented FetaFix, an automated approach for fault localization and repair during model conversion between deep learning frameworks. FetaFix is capable of detecting and fixing six commonly encountered fault types that are input and layer-based related to preprocessing, input dimensions, layer weights, biases, hyperparameters, and computation graph.

We evaluated the effectiveness of FetaFix in fixing previously reported model conversion bugs that we found on our previous work [70] (described in detail in Chapter 3) for three widely used image recognition models converted across four different deep learning frameworks. We found that FetaFix was highly effective in repairing the erroneous model conversions reported by completely fixing or significantly improving 14 out of the 15 cases under test, repairing a total of 462 bugs. We identified the root causes of these bugs and reported them to the development teams of the respective converter tools.

4.8 Availability

The source code, installation instructions and a small test dataset of FetaFix, are available at: https://github.com/luludak/FetaFix.

Chapter 5

Mutation Testing of Models Deployed on Hardware Accelerators

5.1 Introduction

Enabling optimization and hardware acceleration of Deep Neural Networks (DNNs) is vital to achieve high performance [79]. Enabling optimization and hardware acceleration of Deep Neural Networks (DNNs) is vital to achieve high performance. Given the complexity of the task, and in order to automate the process of DNN model compilation, optimization and deployment on different hardware acceleration devices, a number of Machine Learning (ML) compilers have been implemented. OpenAI Triton [73] and Apache TVM [60] allow the compilation and automatic optimization of DNNs, while MLIR [61] provides a low-level framework for developing AI compilers. However, the compilation process can be error-prone, due to the architectural and structural complexity of DNNs, intertwined with effective hardware acceleration orchestration. A recent study [41] has shown that GPU bugs such as incorrect shate sharing and wrong data parallelism on GPUs are among the main faults in DNNs, indicating that the process of integrating hardware acceleration in DNNs needs to be adequately tested. In addition, our prior research indicates that model performance can unexpectedly vary across GPU devices [70, 74].

For that purpose, we propose `MutateNN`, a mutation testing tool aiming to examine the robustness of compiled DNNs in the presence of errors related to hardware acceleration. These errors might be a result of potential AI compiler faults - which are rare, but occur in AI compilation systems [81, 200, 202, 288], or bugs introduced by developers when implementing custom kernels for hardware acceleration. Inspired by mutations generated in conventional software, `MutateNN` is able to generate DNN model mutants, execute them, and compare their behavior to the original model, and can do so across a variety of hardware acceleration devices. `MutateNN` leverages the popular Apache TVM [60] compiler. To demonstrate the capabilities of `MutateNN`,

we selected 7 widely-known image recognition models, which were all pre-trained with the ImageNet [68] dataset. Inspired by the literature and by conventional mutation testing techniques, we generated 21 mutations of 6 different categories: Layer Node Replacement, Arithmetic Node Replacement, Node Input/Output Modifications, Arithmetic Types Mutations, Kernel Variables & Stores Mutations, and Conditional Statement Operations Mutations. Using these mutations, we performed inference on 4 hardware devices of varying computational capabilities. We observed up to 90.3% output label prediction differences across different devices in mutants related to conditional operations an unexpected observation, as the mutation was not anticipated to behave differently across devices, as well as an unexpected model correctness degradation related to mutants of arithmetic types.

In summary, the following contributions are presented in the next sections:

1. A tool (`MutateNN`) that enables mutation testing in DNN models to explore model robustness across different hardware acceleration devices in the presence of faults.
2. An preliminary evaluation that utilizes `MutateNN` to explore the robustness of 7 DNN models against 6 different categories of faults, upon performing deployment and inference on 4 hardware devices (GPUs) of varying capabilities.

5.2 System Architecture

`MutateNN` consists of three main components: the *Model Variants Generator*, the *Execution Module*, and the *Analysis Module*. It utilizes a configuration-based approach, capable of generating, deploying, performing inference and analyzing a batch of model mutations. The system architecture is presented in Figure 5.1. `MutateNN` is built on top of the *TVM* compiler stack [60], which allows parameterization of models and deployment on different hardware acceleration devices. `MutateNN` allows end users to define a configuration of the experiments under test in a JSON file.

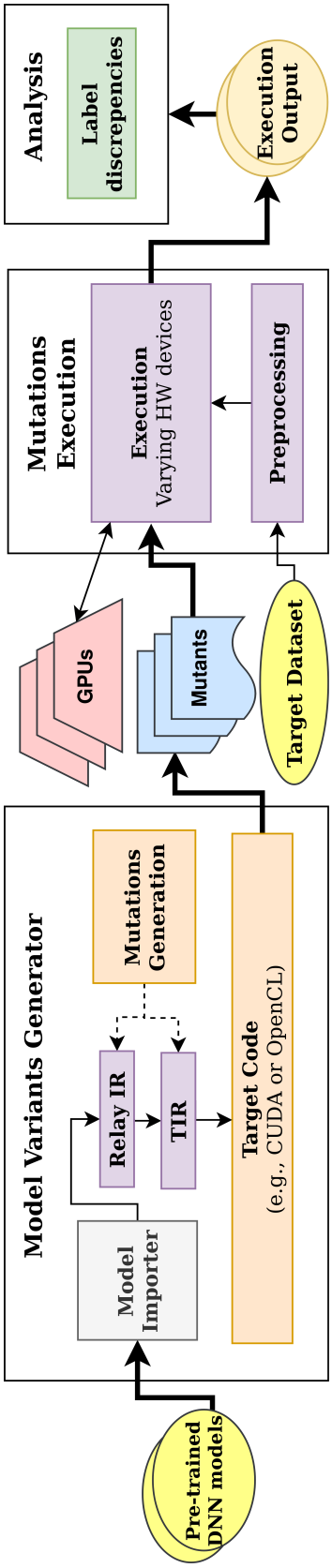


Figure 5.1: Architecture of `MutateNN`: (1) *Model Variants Generator* generates mutations and compiles them to device code; (2) *Mutations Execution* executes the various mutants on images from a target dataset; and (3) *Analysis* compares inference outputs and reports metrics across mutant executions.

Model Variants Generator Module To test the behavior of DNNs across different hardware acceleration devices, `MutateNN` enables the generation of model mutations, focusing on two primary categories: (1) graph-related mutations, implemented at Relay IR (TVM's high-level graph IR); and (2) code-related mutations, implemented at TIR (TVM's low-level IR, one level above device code generation). `MutateNN` loads a model from its initial format, which can be either a pre-trained model from a well-established library (e.g., Keras [47]), or a file-based representation, such as ONNX [58]. `MutateNN` then compiles the mutants following a user-defined configuration by utilizing TVM, generating an executable targeting a specific device. The output is a DNN model in compressed format that can be loaded via TVM, deployed and executed at a later stage in the process. It is important to highlight that the module will only generate valid models, i.e., that succeed through the compilation process of TVM, otherwise a compilation error is raised and the mutant is not further considered for execution.

Mutations Execution Module Once the tool completes the mutant generation process, `MutateNN` loads and deploys a mutant to a hardware acceleration device for model inference, via Remote Process Communication (RPC). Initially, `MutateNN` loads and deploys the model in the hardware acceleration device defined in configuration. Then, it utilizes the evaluation dataset defined in the configuration. In particular, it applies the necessary pre-processing (e.g., normalization) on each dataset image, performs inference against the model mutant, and generates the top-K inference result along with the execution time. Finally, it stores each output to a separate file for further examination by the analysis module. In addition, the execution process is optimized for efficiency, as `MutateNN` progressively verifies each run during execution. It automatically skips runs that result in crashes and terminates execution early if it detects problematic behavior—such as the model producing identical outputs for different inputs.

Analysis Module Once the inference operations are complete for the whole experiment set (the original model and its mutants, executed across all devices using the dataset under test), then `MutateNN` applies analysis to determine discrepancies between mutant and original model executions by utilizing pairwise comparison against the inference outputs generated by different devices for the respective mutant under test. In particular, `MutateNN` loads all files containing the prediction outputs for the dataset, generated for each mutant across devices, and performs pairwise comparison,

using a user-selected metric. Note that MutateNN supports a number of comparison metrics, such as top inference output label comparison, Levenshtein distance and Kendall's Tau. The reporting results are generated in a summary JSON file, enabling further processing.

Configuration MutateNN allows its setup to be configured via a JSON file, which contains properties related to: (1) specification of models under test, along with their properties (e.g., input and output dimensions); (2) mutations to be generated, both in graph and kernel code level; (3) devices where the models will be deployed and executed on; and (4) dataset definitions, utilized for differential testing purposes. A sample of the configuration file, along with directions on how it can be utilized, can be found in MutateNN code repository.

5.3 Implementation

MutateNN is implemented on top of Apache TVM [60], a performance-oriented AI compiler stack that enables DNN optimization and deployment in a variety of hardware acceleration devices. MutateNN utilizes TVM Graph (Relay) and Tensor-Level IR (TIR) to generate model mutants with the purpose of testing the AI compiler-generated code that enables hardware acceleration.

Model Variants Generation MutateNN is implemented so that users can define and configure their own custom mutants, along with their scope and effect range in the DNN model. With regard to Relay, MutateNN slightly mutates parts of the model graph. For instance, it introduces nodes (e.g., transpose), affecting model structure and data manipulation. With relation to TIR, MutateNN applies modifications related to the kernel code (e.g., conditional statements responsible for thread handling), generated from TVM, in order to enable hardware acceleration for the DNN model. We present the set of mutations supported by MutateNN in the next section.

Mutations Supported `MutateNN` supports two primary types of mutations: **(1) Graph-Related**, and **(2) Code-Related**. The mutations are inspired by related work (primarily from Tzer [81] and DeepMutation [86]), by an established taxonomy of real faults in DNNs [41], but also from conventional mutation testing applying changes in expressions and types adapted to the context of generating mutants for target code utilized for hardware acceleration. For graph-related mutations, we generate mutants based on 3 subcategories: **(a) Layer Node Replacement (LN)**, replacing layers and computational nodes (e.g. replacing a `ReLU` node to `Sigmoid`); **(b) Arithmetic Node Replacement (AN)**, replacing nodes related to batch arithmetic operations (e.g., `Add` node to `Subtract`); and (c) **Node Input/Output Modifications (NIO)** (e.g., replacing a data tensor input node to a `Dense` node). It is worth noting that, our mutations list is far from exhaustive. A considerable portion of operators suggested by the literature [11, 81, 86, 113] are yet to be implemented in `MutateNN`. We selected an indicative subset of popular mutations, as a proof of concept for `MutateNN`. In addition, in our runs, `MutateNN` is currently focused in the deployment process, and therefore we disregard any mutations or analysis related to the training process. `MutateNN` allows users to parameterize mutations through its configuration file, enabling customization to fit specific testing needs. Aspects such as the conditional statement operators, constant values in store/variable operations, and layer modifications can be specified in the configuration, which `MutateNN` then uses to automatically generate the corresponding mutants. All graph-related mutations are implemented on Relay IR in Apache TVM. For each subcategory, `MutateNN` traverses the Relay IR Abstract Syntax Tree (AST), and either replaces function calls or injects operations.

In addition, `MutateNN` analyzes model metadata such as input and output dimensions across layers, and adjusts them in order to preserve mutant validity. An indicative example (transposing input tensor of a Conv2D layer) is shown in Figure 5.2. For code-related mutations, we generate mutants based on three subcategories, focusing on operations modifying statements and constructs at a lower level: **(a) Arithmetic Types Mutations (AT)** (e.g., changing `float32` to `float16` in variable types); **(b) Kernel Variables & Stores Mutations (SV)** (e.g., adding a constant of 0.5 to an existing variable holding a numeric value); and **(c) Conditional Statement Operations Mutations (CSO)**, such as replacing the less-than ($<$) operator of a

```
%123 = clip(%122, a_min=0f, a_max=6f) /*Tensor[(1, 384, 14, 14), float32]*/;
%124 = transpose(%123, axes=[0, 1, 3, 2]) /*Tensor[(1, 384, 14, 14), float32]*/;
%125 = nn.conv2d(%124, %onnx::Conv_564, padding=[0, 0, 0, 0], kernel_size=[1, 1])
```

Figure 5.2: Injected tensor transposition in Relay IR.

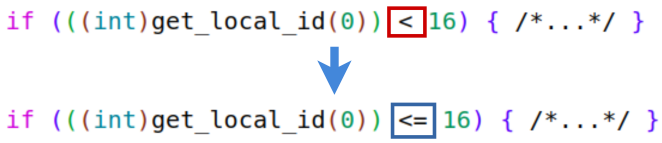


Figure 5.3: Mutation of a conditional operator in the device kernel code (OpenCL) for a fused operation in MobileNetV2.

test expression inside a conditional statement by less-than-equal ($<=$), as shown in Figure 5.3. These mutations are applied at the lower-level, Tensor IR (TIR) of Apache TVM. `MutateNN` traverses the TIR AST, and mutates the necessary statements and expressions based on configuration.

Inference Analysis `MutateNN` is capable of automatically analyzing and comparing the execution results of varying mutants, by identifying the execution results of each mutant and comparing them using pairwise comparison. For instance, consider MobileNetV2 mutated to *MobileNetV2M1* and the mutant was configured to run on 3 devices (*D1*, *D2*, and *D3*) against a dataset defined in configuration. Following execution on all the above configurations, `MutateNN` will identify the folders with the outputs and compare, element-wise for each dataset output, across all combinations of devices (*D1D2*, *D1D3*, *D2D3*). `MutateNN` supports a variety of element-wise comparator strategies, such as top-1 prediction, Levenshtein distance, and Kendall's Tau [289]. In addition, `MutateNN` performs the comparisons efficiently, by automatically ruling out redundant comparison cases, such as where a mutant crashed, or was stopped early from the Mutations Execution Module, because it produced unreasonable results (e.g., all inferences having the same or very similar predictions) for one or more devices (as described in Section 5.2). In that scenario, `MutateNN` will disregard the case and proceed only with the rest of the valid runs. As a result of the analysis, `MutateNN` generates a number of JSON files containing metadata, such as the difference percentage across output label predictions and the dataset image output names, that presented discrepancies.

5.4 Evaluation

We considered 7 widely-used image recognition models of varying sizes and architectures, used for image classification, as well as more complex activities, such as semantic segmentation and object detection: ShuffleNet [290] (5.46MB, 2M parameters), MobileNetV2 [71] (14MB, 3.4M parameters), ResNet152V2 [67] (230MB, 115.6M parameters), AlexNet [230] (233MB, 60M parameters), EfficientNet (Lite) [291] (49.5MB, 5.3M parameters), DenseNet121 [292] (31.2MB, 8M parameters), and InceptionV2 [221] (43MB, 13.6M parameters). All models are pre-trained on ImageNet [68]. We use the object detection test dataset of the ImageNet Large Scale Visual Recognition Challenge [69] as our experiments base dataset, given its complexity and realistic content. We tested our original DNN models and their respective mutants against 4 hardware accelerators of different levels of capabilities (the same devices utilized for experiments in Chapter 3):

- an Intel-based server featuring an Nvidia Tesla K40c (GK11BGL) GPU (*Server*), with a computational power of 4.29-5.04 TFLOPS for FP32 computations [224, 225],
- a Nvidia AGX Xavier featuring an Nvidia Volta GPU (*Xavier*), with a theoretical processing power of 32 TOPS and a calculated, theoretical performance of ≈ 1.4 TFLOPS [226],
- a Laptop with an i5-8365U CPU @ 1.6GHZx8, featuring an integrated Intel(R) GEN9 HD Graphics NEO (*Local*) with 24 Execution Units, and therefore a performance of ≈ 307 GFLOPS for FP32 operations [227],
- and a mobile-class Hikey 970 board featuring an Arm Mali-G72 GPU (*Hikey*), with a theoretical processing power of 244.8 GFLOPS for FP32 operations [228].

Initially, we run the original models across devices to check for potential problems without applying any changes to them, and we detected no differences in output predictions. We then proceeded generating mutations and performing differential testing across devices. Overall, we generated 21 mutations covering all 6 categories supported by `MutateNN`, as described in Section 5.3. Our mutation set is presented in Table 5.1.

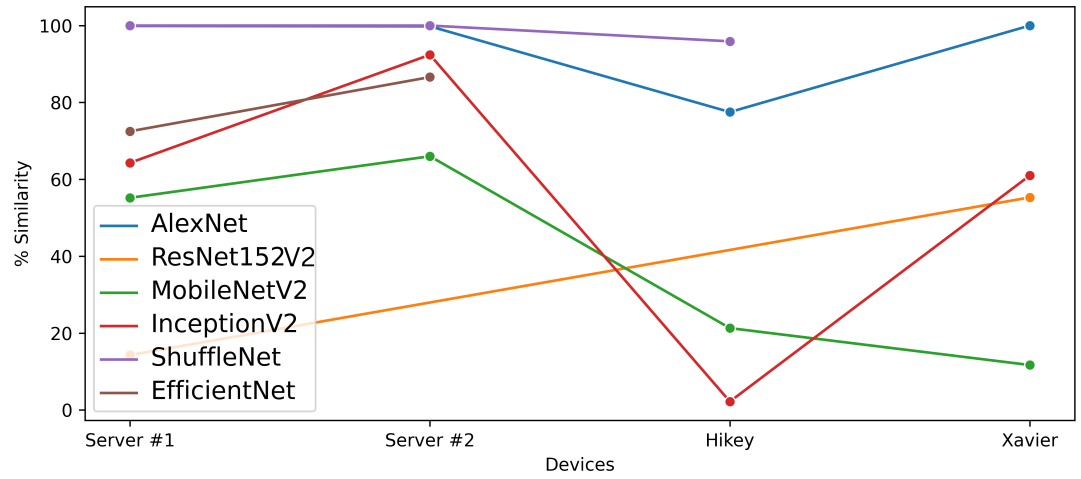
Table 5.1: Mutations generated for evaluating `MutateNN`.

Cat.	Mutations
LN	ReLU $\rightarrow$ Sigmoid
AN	Add $\rightarrow$ Subtract
NIO	Transpose(Bias) for *Conv2D*, Transpose(Input) and Exp(Input/Output) for *Conv2D*, *Dense*, *BatchNorm*
AT	Float32 to Float16, Int16, Int8
CSO	LT($<$) $\rightarrow$ LTE($<=$), GT($>$) $\rightarrow$ GTE($>=$), GT($>$) $\rightarrow$ LT($<$), $x < y \rightarrow x < (y + 0.5)$
SV	Var/Store Sub(1e - 8), Mul(1 + 1e - 8)

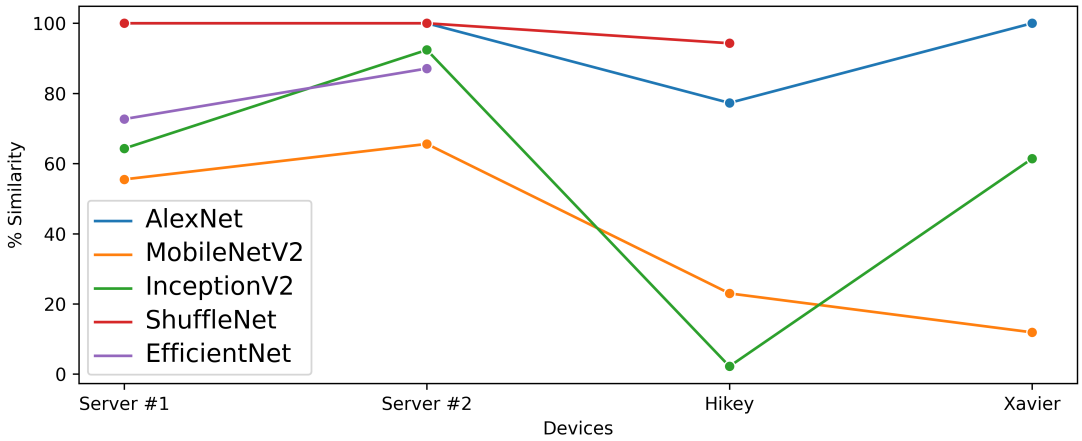
We present our results in the sections 5.4.1 and 5.4.2 below. Our focus is primarily on the observation of model behavior differences across devices as a result of the fault present on the corresponding mutant, and not on cases where mutants resulted in identical behavior across devices. This is because essentially a mutant that is consistently problematic can identified as problematic and "killed" (either by testers manually, or from a potential testing suite) with ease, while for cases where the behavior is inconsistent across devices, the effects of a fault has a much higher chance to be overlooked. The only exception to the rule is with regard to *AT* category of mutants, where the model behavior was unexpected, given the nature of the fault introduced - and which we present in more detail.

5.4.1 Graph-Related Mutants

For graph-related mutants, we concluded with the following observations: **LN:** The replacement of *ReLU* by *Sigmoid* occurences in mutants led most of the cases to crash, with the exception of *EfficientNet*. *Xavier* device demonstrated robustness to this change, while deploying the mutant to the other tested hardware acceleration devices resulted in a crash. **AN:** We observed heavy model accuracy degradation (up to 99.8% on *InceptionV2*) that was consistent across devices. **NIO:** The transposition of both data and weights of convolutional nodes (Conv2D) led to heavy model accuracy degradation (up to 98.4% on *InceptionV2*), a behavior that was also consistent across devices.



(a) LT-to-LE



(b) LT-RA

Figure 5.4: Comparison of mutants against the original model across devices: (a) LT to LTE and (b) right value increased by 0.5 in conditional statements for all the DNNs under test.

5.4.2 Code-Related Mutants

For code-related mutants, the results were the following: **AT:** The models produced predictions that instead of considering the full set of classification labels, they were limited to a very small subset of them ($< 2\%$), for all the dataset inputs, misclassifying the vast majority of them. For example, for *EfficientNet*, the mutant of changing arithmetic types from *Float32* to *Int16*, distinct images containing a goldfish and an owl, are both mistakenly identified as "paddle wheel", essentially resulting in a not crashed, but practically unusable model. Although this result was consistent across devices, we considered it interesting, as many optimizations (e.g., fast-math) compromise model

accuracy over computation speed and size. Furthermore, a change in arithmetic types (e.g., *Float32* to *Int16*, was expected to produce an effect related to only partial accuracy drop, due to loss of precision, a considerably different behavior in comparison to the one observed in our experiments with *AT* mutants, as aforementioned. **SV:** The mutations seemed to have no effect to the model correctness. However, our constant mutations in store values were limited in the models. We consider applying more drastic changes in future work. **CSO:** we observed different behavior across hardware acceleration devices in cases of (1) operator modifications and (2) changes in conditional thresholds. For (1), and although most of the conditional operator mutants led to similar results across devices, the mutation changing the less-than (LT) to less-than-equal (LTE) operator in conditional statements (LT-to-LE) produced up to 90.3% discrepancy, observed in InceptionV2 across *Server #2* and *Hikey*. For (2), an increase of the right operand value of a conditional statement using LT by a constant (LT-RA) presented different behavior across devices, with once again *InceptionV2* presenting a difference of up to 90.2% across the same devices with (LT-to-LE). Mutations related to the greater-than (GT) operator had no effects to the models across devices, with the exception of *EfficientNet*, which crashed on *Hikey*. The results of both mutants can be observed in Figure 5.4, following a similar trend, as both of them essentially modify the lower bound of a LT condition. For the rest of the mutants related to conditional operators, no significant results were observed.

5.4.3 Results Overview

Overall, we conclude to the following observations: (1) Differences in kernel code related to conditional operations can affect output label correctness across devices, in varying degrees. (2) Heavy modifications in model types associated with precision change in kernel code (*AT* mutants) can lead to severe model performance degradation, resulting the models giving similar output classifications for completely unrelated inputs, resulting in practically unusable models. (3) For a wide variety of graph and kernel-related faults introduced, the behavior is consistent across devices (including cases where mutants crashed across devices). Observation #2 also falls into this category, however it results in an unexpected, peculiar behavior across the models. Furthermore, we consider it as a separate, noteworthy observation. Both #1

and #2 observations can be attributed to the differences in implementation across programming models and GPU architectures [293]. However, exploring and identifying precisely the aspects of the observed behavior across each specific device, is out of the scope of this work.

5.5 Tool Usability

`MutateNN` allows users to define a JSON-based configuration, containing all the DNN models under test, a set of mutations related to the model graph and GPU kernel code, as well as the hardware acceleration device configuration that the compiled models will be deployed on, while utilizing different datasets in order to apply pairwise differential testing of all mutants deployed across devices. Users can use `MutateNN` by having only basic knowledge on DNN deployment. `MutateNN` is able to automatically fetch a plethora of pre-trained models from well-established libraries, such as PyTorch [46] and Keras [47], while also supports file-based custom model representations, such as ONNX [58]. In addition, since `MutateNN` is built on top of Apache TVM, a generic purpose ML compiler, it theoretically supports mutation testing on DNN models outside the image recognition domain, such as text classification. Related to results interpretability, `MutateNN` supports differential testing across execution configurations in a pairwise manner, generating rich metadata reports in JSON format, both human and machine-readable. Finally, the mutation strategies in `MutateNN` can be applied to other AI compilers and frameworks, such as MLIR [61].

In summary, `MutateNN` provides a highly configurable and extensible mechanism in order to apply differential testing across hardware acceleration devices. Furthermore, we propose the following use cases for `MutateNN`. **(1) Mutants Generation to Examine Test Suite Effectiveness:** `MutateNN` can be used as a generator of mutants so that effectiveness of test suites written with the purpose of ensuring robustness of models across different hardware acceleration devices. Following the traditional mutation testing paradigm `MutateNN` can generate mutants that examine DNN model behavior in the context of hardware acceleration. **(2) Simulation of Errors to Examine Model Behavior:** `MutateNN` can be used to examine how DNNs behave in the presence of undesired faults. For instance, imagine that a conditional statement error in the kernel of a model is introduced by a developer, and this model is compiled and deployed as part of two models of an autonomous driving system that use the same model, but different sets GPUs for calculations.

MutateNN, can be used from both DNN model developers and testers can simulate the effects of errors across different hardware acceleration devices, collect metrics and observe the severity of impact that such errors have. This can help them develop fail-safe mechanisms that prevent catastrophic scenarios. In addition, since MutateNN is a configuration-based utility that automatically compiles, deploys, and evaluates models, it can be integrated into the automated Continuous Integration and Deployment (CI/CD) pipeline of the model development process, contributing to model robustness from the development phase.

5.6 Discussion

From our findings we can infer the following considerations: **(1) Differences in Conditional Operators:** Conditional operators in kernels are quite commonly used to indicate thread computations grouping. A quite common problem in conventional software kernels is branch divergence [294], which mostly affects parallelization efficiency. However, our observations highlight that problematic conditional statements can also affect the predictions of the DNN model across different devices in a drastic manner and to a varying extent. For example, Fig. 5.4a demonstrates that an LT-to-LE mutant generated for *AlexNet* run on a device like *Server #1*, achieves results of $\approx 100\%$ accuracy compared to the original. However, the accuracy of the same mutant deployed on *Hikey* drops to 77.5%. Similarly, for *Shufflenet*, the same mutant gives 100% accuracy compared to the original on *Server #1*, but 95.85% on *Hikey*. If a test suite used to check model correctness allowed an accuracy deviation threshold of just 1%, and the models were not tested on low-end devices such as *Hikey*, both mutants would probably survive. As a conclusion, we recommend that model developers extensively test for such defects across a plethora of devices, while enforcing strict thresholds in terms of acceptable accuracy deviations across devices.

(2) Effects of Numeric Type Precision: Usually, precision errors are expected to affect model correctness. For instance, quantization [295, 296] aims to improve the computation times of a model at an acceptable cost of model accuracy. However, we observed that such modifications, when applied in large scale, can affect the model so drastically that they can turn it essentially unusable. As a result, we suggest to the DNN model developers to be cautious when aiming to deploy models for hardware acceleration, as modifications that are related to changes in numeric precision for

purposes such as optimization (e.g., quantization [297]) can potentially have a much more adverse effect to model accuracy than the initially anticipated. Furthermore, model developers should test the models to this direction across hardware acceleration devices.

As a result, we suggest to the model developers to be cautious when aiming to deploy models for hardware acceleration, as bugs that are expected to have a limited adverse effect might affect it severely. To this direction, we recommend the definition of tests that explicitly focus on the DNN model behavior across devices, and consider conditional operators and numeric precision types differences.

5.7 Availability

The source code of `MutateNN` can be found at `www.github.com/luludak/MutateNN`. A tool demonstration can be found at `www.youtube.com/watch?v=j7Ffd5y_i3g`.

5.8 Summary

We presented `MutateNN`, a tool that enables mutation testing of DNN image recognition models in the context of hardware-accelerated deployment. `MutateNN` can be used both to assess the effectiveness of a DNN model suite in the context of hardware acceleration, and to simulate faults related to problematic target code. We demonstrated the usage of `MutateNN` by generating 21 mutations for 7 DNN models and executed them in 4 hardware acceleration devices of varying capabilities. We observed up to 90.3% output label deviations in mutants related primarily to threshold approximations and conditional operators, as well as a number of unexpected severe model performance degradation cases, related to arithmetic type mutations.

Chapter 6
Conclusion

In this chapter, we present a summary of the contributions of this thesis, reflecting on the impact of the work, presenting its limitations, while also proposing potential directions for future extension and improvements.

6.1 Summary of Contributions

This thesis presents three main contributions related to testing of Deep Neural Networks (DNNs): (1) `DeltaNN` a suite enabling differential testing in order to assess the impact of computational environment parameters on the accuracy and execution time performance of DNNs (with emphasis on image recognition models) in the context of model deployment; (2) `FetaFix`, a suite for automatic fault localization and repair of faulty Deep Learning framework conversions; and (3) `MutateNN`, a suite for mutation testing to test DNN model robustness against miscompilations related to target code utilized for hardware acceleration.

To the best of our knowledge, all works are novel and attempt to address three problems found in the literature and reported in existing issues in related Q&A websites, forums and code base issue trackers, which have been overlooked by the literature.

In addition, all our works incorporate the effective usage of software configuration [298, 299] and variability [300], allowing users to adapt the systems to their specific needs. We enable users of all our systems to define a set of configuration attributes and constraints in a system configuration file—ranging from metadata related to models and datasets, to device deployment information, optimization pass selections (in `DeltaNN`), fault localization and repair configurations (in `FetaFix`), and mutant definitions (in `MutateNN`). Our systems can be configured accordingly and are designed to generalize beyond the scope of our experimental setups.

Since we have made all our tool codebases publicly available, we hope they will be used by developers and other researchers to address their own needs and to serve as foundations for novel research. Furthermore, our systems are built to be easily extensible, supporting new types of models and workflows tailored to user requirements. For example, users can perform fault localization and repair on object detection models using `FetaFix` by providing an `Evaluator` comparator class, similar to how classification models are currently handled in the system. This principle of variability is embedded across all our systems, which we designed with extensibility and ease of use in mind.

6.1.1 Computational Environment Impact to Image Recognition Models

The effects of the computational environment on the DNN model deployment process is generally overlooked in the literature. Existing techniques do not consider DNN model errors that could potentially be caused by interactions of the model with aspects such as conversions between Deep Learning (DL) frameworks (e.g., TensorFlow and PyTorch), compiler optimizations (e.g., operator fusion, loop unrolling, etc.), and the hardware platforms they run on (e.g., CPUs, GPUs, etc.).

To address this, we presented `DeltaNN`, a tool that helps evaluate the robustness of image recognition models to changes in specific aspects of the computational environment. Using `DeltaNN`, we assessed the robustness of DNN models with respect to (1) output label correctness and (2) model inference time performance (crucial for timing safety in real-time perception systems in applications like self-driving cars). We conduct experiments on three widely used image recognition models (MobileNetV2, ResNet101V2 and InceptionV3) by (1) converting them across 4 DL frameworks (TensorFlow, TF Lite, Keras, PyTorch) (2) applying 3 different compiler optimization scenarios and (3) deploying across 4 hardware acceleration devices of varying capabilities. For each of the configurations aforementioned, we conducted differential testing using the object detection test dataset from ImageNet.

Overall, we observed that conversions between DL frameworks significantly impacts output labels of the DNN models by up to 100%. In addition, we observed that varying hardware accelerators and compiler optimizations do not affect model output accuracy, but can lead to a non-negligible execution time performance degradation with respect to inference time under specific scenarios. In particular, we observed up

to 81% unexpected execution time performance degradation in model inference times when applying certain compiler optimizations and deploying the optimized models to low-end hardware acceleration devices, while the same optimizations improve the execution time performance of the model on high-end devices.

6.1.2 Automatic Fault Localization and Repair of Model Conversions

With the widespread use of DL in various domains, there is an inherent need for DL models to be inter-operable across DL frameworks (such as PyTorch and Keras) to maximize re-usability and extensibility of models across frameworks. Conversion of DL models between frameworks is facilitated by a plethora of automated conversion tools (e.g., `tf2onnx` and `MMdnn`). However, the conversion process may be riddled with bugs, making the converted models undeployable, perform poorly in terms of output label correctness when changing from one framework to another, run slowly, or face challenges in robust deployment.

To address this problem, we proposed `FetaFix`, an automated approach for fault localization and repair of erroneous model conversions between DL frameworks. We focus on the error cases in Figure 3.4 with non-zero output label differences. `FetaFix` is capable of detecting and fixing faults introduced in the layer weights, biases, hyperparameters, and the model computation graph of the converted *Target* model, as a side effect of the conversion process. We evaluate the effectiveness of `FetaFix` in fixing the model conversion errors we observed in Chapter 3. `FetaFix` was found to be highly effective towards fault localization and repair, being able to effectively localize and repair 14 of the 15 erroneous model conversions that had non-zero output label dissimilarities, as well as identify as actual faults and fix 462 out of 755 detected differences across *Source* and *Target* models.

Our methodology is inspired by conventional differential testing [104], specifically adapted for DNNs. By comparing the behaviors of two model variants (*Source* and *Target*) our approach efficiently identifies and localizes faults through a greedy fault localization strategy. This methodology enables the detection of accuracy deviations, result of a problematic DL framework conversion process.

A key feature of our approach is its integration of delta debugging principles [238], which aims to isolate the minimal set of DNN model aspects and attributes responsible for accuracy drops in the *Target* model in relation to *Source*. This provides a clear understanding of the conversion issues that impact the prediction accuracy of

Target. The iterative and search-based nature of our methodology on layer-based fault localization strategies, provides a scalable and effective way to pinpoint discrepancies between models, offering a divide-and-conquer approach that isolates the layers with the most significant impact on accuracy.

The incorporation of Search-Based Fault Localization (SBFL) [239, 240] further strengthens our process. We adapt it for DNNs to precisely identify and rank suspicious layers, improving fault localization. Our method combines one-off repairs with iterative SBFL strategies, progressively refining *Target*. Repairs are validated on sample inputs and, if effective, applied to the full dataset, continuing until *Target* is fully repaired or no further improvements are possible.

Overall, the methodology presented here offers a comprehensive, adaptive approach to fault detection and repair in deep neural networks. By combining differential testing, delta debugging, and SBFL, we have demonstrated a powerful tool for improving model robustness and accuracy in a systematic and efficient manner.

6.1.3 Mutation Testing of Models Deployed on Hardware Accelerators

Enabling optimization and hardware acceleration of DNNs is vital to achieve high inference time performance. For that purpose, a number of ML compilers have been implemented (e.g., Apache TVM and OpenAI Triton). However, the compilation process can be error-prone, due to the architectural and structural complexity of DNNs, intertwined with effective hardware acceleration orchestration and needs to be adequately tested.

For that purpose, we propose `MutateNN`, a mutation testing tool aiming to examine the robustness of compiled DNNs in the presence of hardware-related faults potentially introduced by issues in the AI compilers or bugs introduced by developers when writing custom kernels for hardware acceleration. Inspired by mutations generated in conventional software, `MutateNN` is able to generate DNN model mutants, execute them, and compare their behavior to the original model, and can do so across a variety of hardware acceleration devices.

To demonstrate the capabilities of `MutateNN`, we selected 7 widely-known image recognition models, which were all pre-trained with the ImageNet dataset. Overall, we generated 21 mutations of 6 different categories: Layer Node Replacement, Arithmetic Node Replacement, Node Input/Output Modifications, Arithmetic Types Mutations, Kernel Variables & Stores Mutations, and Conditional Statement Operations Muta-

tions. Using these mutations, we performed inference on 4 hardware devices of varying computational capabilities. We observed up to 90.3% output label prediction differences across mutants related to conditional operations, as well as an unexpected model correctness degradation related to mutants of arithmetic types.

Combining mutation testing with differential testing in `MutateNN` helps examine DNN model robustness in the presence of hardware-related faults. This approach simulates problematic scenarios to assess model performance and test suite effectiveness across different hardware acceleration devices. In doing so, `MutateNN` provides a powerful method to contribute to the development of robust and reliable DNN models, particularly in ensuring correctness across various hardware platforms.

6.2 Critical Reflection

Through the contributions proposed in this thesis, we learned the following lessons:

- DNNs are error prone to computational environment aspects. Overall, (1) DNN model correctness was affected when utilizing different DL frameworks in order to implement them, (2) compiler optimizations did not affect model correctness but behaved in an inconsistent manner across hardware acceleration devices of varying capabilities, and (3) GPU devices produced consistent results across devices, but (as expected) affected model execution times, but not always in an anticipated manner (as (2) suggests).

- Model conversions across DL frameworks is an error-prone process. We discovered 6 potential causes that hinder the process and effectively performed fault localization and repair for conversions across 4 DL frameworks applied in 3 popular classifiation models used for image recognition.

- DNN models are generally robust in the presence of faults such as kernel variable value modification and changes in the model graph structure and its related operations, in the context of hardware acceleration, however issues related to conditional expressions can affect model behavior across different devices.

6.3 Limitations and Future Work

In this section, we present limitations in our work, as well as potential directions of improvement and expansion of our work. In total, we propose three directions: (1) extension of work towards different DNN types, (2) automatic repair of crashed DNN model conversions, and (3) coverage-guided mutation testing.

6.3.1 Limitations

While our work makes significant contributions, it also poses certain limitations:

- Our current implementations across all of our contributions, support the evaluation of image recognition models performing classification tasks. Although our methodology is solid, and expansion of this work to be applicable to models of other categories and more complex architectures is theoretically straightforward, it is an aspect not currently integrated in our tools. This would require the implementation of comparator classes that are context-specific to the models under test (e.g., comparison of bounding boxes in models performing object detection tasks).

- Our tool that localizes and repairs faults in converted DNN models (`FetaFix`) has certain limitations: (1) `FetaFix` cannot repair models whose structure is so drastically changed through the conversion process that layer matching is not possible, (2) due to inherent limitations related to graph transformations in other model formats (e.g., TFLite not allowing changes on the model graph extracted in `.tflite` format), `FetaFix` primarily performs model repair at the ONNX format, (3) `FetaFix` does not support repair of converted models that crash upon execution, and (4) the layer analysis process included in `FetaFix` is resource intensive, which can be a limitation, and in fact unprofitable if used in models of small size, where a greedy repair approach might be proven more effective. For this reason, `FetaFix` allows fault repair process both with layer analysis enabled and disabled, following a greedy approach on the latter case.

- Our mutation testing tool (`MutateNN`) does not support coverage-guided testing. Although it is built with the purpose of integrating such feature, this needs to be added and is subject to future work.

6.3.2 Extension of Work Towards Different DNN Types

In this thesis, we conducted experiments on CNNs utilized for image recognition, performing image classification tasks. The selection of such models was done for tractability purposes, as they are generally manageable in size, limited in terms of resource demands, and easier to evaluate, in real-world scenarios. However, our methodologies can theoretically be applied to either image recognition models performing more complex tasks (e.g., object detection and semantic segmentation), or models of different families (e.g., Long Short-Term Memory (LSTM) [301] Models and Transformers [209]).

In the context of differential testing, the DNN models should be compiled in `DeltaNN` without problems, as `DeltaNN` is built on top of Apache TVM and well-known converters. However, a comparator class should be defined, so that differential testing can be conducted. For example, a CNN conducting object detection should be able to compare which results are considered "same" or "different" across model inference, including parameters such as the number of objects identified, the bounding box values and data, etc. Another interesting direction would be towards investigating the effects of the computational environment on the text output of Transformers utilized for content generation purposes (e.g., Large Language Models).

In terms of fault localization and repair of DL conversions, another crucial aspect must be set: the definition of a core computational nodes and parameters under comparison for the type of the model. Given that CNNs (all our models in our experiments) utilize convolutions, we utilize the set of convolutions as core computational nodes and compare its weights and biases in order to identify common layers across *Source* and *Target* converted models, rank them and apply repairs. For models not containing convolutions (e.g., RNNs), a different set of core nodes must be defined. This is subject to future work - to explore which node types could potentially be effective for fault localization and repair in different DNN types.

6.3.3 Repair Crashed DNN Model Conversions

Our work related to repairing DNN model conversions focuses primarily on cases that the conversion process completed successfully, but the model correctness was degraded. A potential extension of our work can be towards identifying the common causes that cause model conversion crashes, and implement automatic fault localization and repair strategies to make them successful - similarly with the methodology we followed in `FetaFix` for non-crashing conversions.

6.3.4 Coverage-Guided Mutation Testing

We followed an arbitrary and greedy mutation testing approach towards exploring the robustness of DNN models when faults are present, in the context of hardware acceleration. Essentially, we applied mutations across models on every operator occurrence under test on each scenario, to magnify the potential effect of a fault and observe the worst-case scenario. However, it would be interesting to restrict the amount of mutations applied within a model based on coverage-guided metrics, as this approach would be more realistic to happen in a real-world scenarios. `MutateNN` supports mutating specific occurrences of operators, but integration with coverage (e.g., neuron coverage) needs to be implemented in the system. In addition, `MutateNN` could benefit from a search-based approach in which the mutations defined in its configuration are incrementally applied in multiple steps, with their effects evaluated after each stage. This would enable `MutateNN` to provide more precise insights into the impact of faults, more closely resembling the behavior of real-world issues.

6.4 Concluding Remarks

Our contributions highlight that the computational environment can have an adverse effect towards the correctness and the inference time performance of DNNs. Commonly used procedures, such as conversions of DNN models across DL frameworks, which should work without issues in theory, are proven prone to errors in practice. We advise DNN developers and researchers to: (1) not take converter correctness for granted; (2) not take the converted model correctness for granted, even if the conversion process appeared to complete successfully, or a small test set of inputs appears to work well in the model, but to further explore its correctness by performing a larger amount of tests before using them for production purposes. In addition, we recommend that DNN developers should not take optimizations for granted - as an optimization might not be an execution time performance panacea across hardware acceleration devices with different types and computational capabilities. We showed that optimizations that worked positively for high-end hardware acceleration devices, resulted in a severe performance degradation in lower-end devices. Finally, we showed that hardware-related target code faults related to conditional statements, can lead to a variety of effects across different hardware acceleration devices, and that arithmetic

types can also lead in unexpected, severe penalty towards DNN model correctness. Once again, developers and researchers should not take for granted that a compiler will successfully perform its task in generating target code across a variety of devices, and be extremely cautious towards errors related with conditional statements.

Appendix A

Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models

This is the supplementary material of our contributions, with regard to assessing the impact of computational environment parameters of image recognition models, as investigated using `DeltaNN` in Chapter 3.

A.1 Important Notes

- For ResNet101 sourced from PyTorch, we selected the V1 version the model instead of V2 as the V2 version was not provided in the official PyTorch repository.The version difference may have a larger effect on model inference time when we compare across DL frameworks, presenting large differences. We therefore ask the readers to take this into consideration for results involving ResNet101 sensitivity to DL framework version differences (Figures A.5d, A.6d, and A.7d).
- MobileNetV2 crashed upon execution on `Xavier` device. Furthermore, there are no plots related to it in Figures A.2-A.4.
- InceptionV3 failed to compile on `Keras`. Furthermore, we ruled out the comparison from Figures A.8-A.10.

A.2 Output Label Correctness across DNN Frameworks

Existing work [161, 162] has already explored and highlighted differences across DL frameworks, in terms of output label correctness. We also conducted experiments to this direction in terms of comprehension, and found some differences. In particular, we present results for the four models under study in Figure A.1, with the TVM compiler optimization level set to Default (-o2), and the hardware acceleration device set to `Server`.

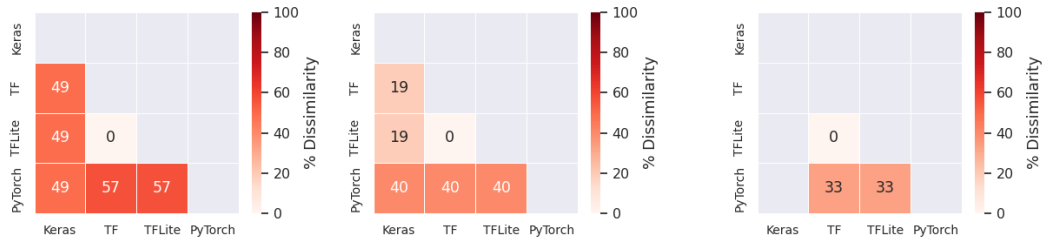
We then vary the DL frameworks one at a time to compute sensitivity of model output label to that framework. Figures A.1a–A.1c show that models are acutely sensitive to the DL framework they are sourced from. Changes in the framework has a significant impact on output label, with MobileNetV2 exhibiting most discrepancy in output labels, in the range of 49-57% for different DL frameworks. We analyse each model below:

MobileNetV2 (Figure A.1a) we observe a 49% dissimilarity between Keras versus the other three frameworks and a 57% dissimilarity for PyTorch versus TF and TFLite. We hypothesize that lower complexity and size of MobileNetV2 makes it less robust to changes in the framework.

ResNet101V2 (Figure A.1b) Keras has a 19% dissimilarity against TF and TFLite. PyTorch resulted in a 40% dissimilarity against all the other frameworks.

InceptionV3 (Figure A.1c) we observe a 33% discrepancy between PyTorch versus both TF and TFLite. No discrepancies were observed for TF versus TFLite; the same was true with other models. Unfortunately, we were unable to load the Keras model for InceptionV3, as it failed to load with our configurations.

Overall, TF and TFLite models always produce the same output, which suggests that the official TFLite models are successfully converted TF models, and contain the same parameters. This is further confirmed by their achieving the same accuracy as seen in Table 2.1. However, when comparing against Keras and PyTorch, we observe significant differences. The extent of difference varies widely between the models, with MobileNetV2 being more sensitive in changes to the DL framework (see Figure A.1a).



(a) MobileNetV2 **(b)** ResNet101 **(c)** InceptionV3

Figure A.1: Pairwise comparison of output label dissimilarities (%) between DL frameworks for our 3 models, running on `Server`, on *Default* optimization level.

For the reason that such differences have already been indicated in the literature, in our main contribution we focus another crucial aspect related to DL frameworks: conversions from a DL framework to another - a common practice applied by developers and scientists, for reasons of model compatibility and extensibility.

A.3 Execution Times across DNN Frameworks

As part of our experiments, we measured the execution times across DNN frameworks and compared them. The results are shown in Figures A.2-A.10. We summarize the following observations with regard to the results:

- *InceptionV3* appears the most robust across all devices, for all the DNN frameworks (except *Keras* which did not run), posing a maximum (small, but non-negligible) relative change of 8% (*TF*/*TFLite* to *Keras* configuration, `Hikey` device).
- *MobileNetV2* appears to be the less robust configuration, with a wide range of deviations across devices. Apart from the *PyTorch* configuration, `Xavier` failed to execute, therefore we do not present any comparisons for that device.
- Considerable deviations were observed in the devices that run the experiments, especially between *Keras* and *PyTorch* (maximum relative change of 16-18% on (`Server`) across optimizations).

- *ResNet101V2* was proven robust for the majority of its DNN Framework configurations, however large discrepancies were observed for its PyTorch configuration. We consider that the *V1* DNN Framework backend used can pose significant differences between the other configurations - going out of range in Figures A.5-A.7 in the last column. Overall, the relative change goes up to 100% for *PyTorch* in comparison to other DNN Framework configurations.

In addition, we observed that *MobileNet* was much less robust than the other models across optimizations. Finally, we observed a wide range of relative changes of the Hikey values across all DNNs, as one can observe in their Interquartile Range (IQR) in box plots.

A.4 Execution Times Across Devices

We reiterate that we used the following devices for our experiments, in Chapter 3:

- an Intel-based server featuring an Nvidia Tesla K40c (GK11BGL) GPU (*Server*), with a computational power of 4.29-5.04 TFLOPS for FP32 computations [224, 225],
- a Nvidia AGX Xavier featuring an Nvidia Volta GPU (*Xavier*), with a theoretical processing power of 32 TOPS and a calculated, theoretical performance of ≈ 1.4 TFLOPS [226],
- a Laptop with an i5-8365U CPU @ 1.6GHZx8, featuring an integrated Intel(R) GEN9 HD Graphics NEO (*Local*) with 24 Execution Units, and therefore a performance of ≈ 307 GFLOPS for FP32 operations [227],
- and a mobile-class Hikey 970 board featuring an Arm Mali-G72 GPU (*Hikey*), with a theoretical processing power of 244.8 GFLOPS for FP32 operations [228].

We present the execution times for *PyTorch*, as shown in Figures A.11-A.13:

- In general, we observed a considerable range of execution times in terms of percentage change, with the largest one presented on *ResNet101V2*, across `Server` and `Hikey` ($773.87\times$) when applying *Extended* optimization. However, this is expected to an extent as capabilities of the devices are considerable. For instance, `Server` is expected to achieve $\approx 20.5\times$ more FLOPS than `Hikey` on a theoretical level (a difference that might be found considerably larger in practice).

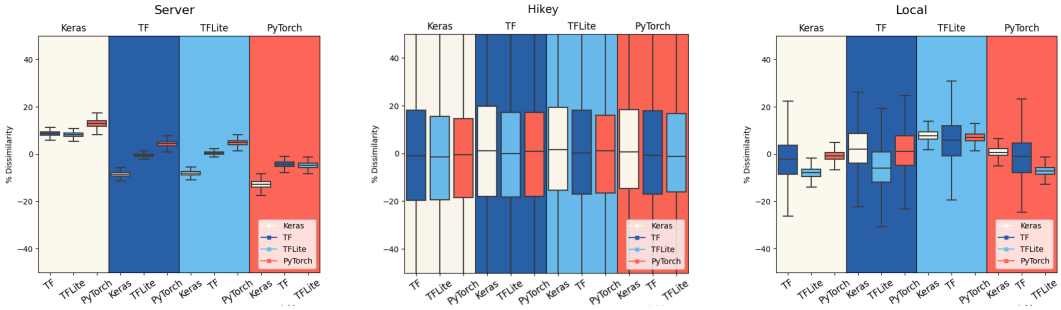
- For *MobileNetV2*, execution time relative change increased by $4.7\times$ between `Server` and `Hikey`, across *Default* (8085%) and *Extended* (38692%) optimizations, posing it the less robust model across this comparison metric as well.

- Also, ResNet101 posed a $3.4\times$ increase across `Server` and `Hikey` for the same range of optimizations (*Default* was 22719%, while *Extended*, 77387%), posing it as considerably sensitive as well.

- However, the inference time performance differences across optimizations does not follow the same norm across models. For instance, the performance comparison of `Server` and `Hikey` appears to increase in `MobileNetV2` (from $\approx 7\times$ using *Basic* to $\approx 8\times$ using *Default* and then $\approx 38\times$ using *Extended* optimizations). The same optimizations lead in flunctuationg performance differences in `ResNet101V2`, ($\approx 36.7\times$ using *Basic* to $\approx 22.7\times$ using *Default* and $\approx 77.3\times$ using *Extended* optimizations), while in `InceptionV3`, the differences remained to about the same levels across all optimizations ($\approx 51.7\times$ using *Basic* to $\approx 56.5\times$ using *Default* and then $\approx 57.9\times$ using *Extended* optimizations).

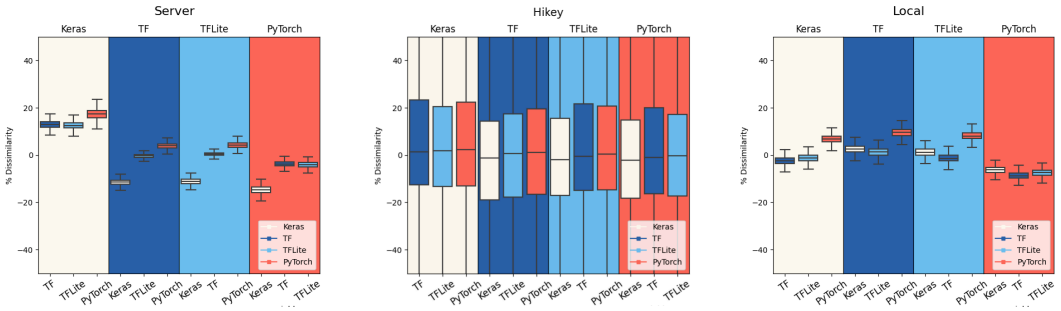
- Additinally, we observe that the differences across devices present deviations across the models. For instance, `MobileNetV2` presented up to $\approx 38.6\times$ differences between `Server` and `Hikey`, while `ResNet101V2` presented up to $\approx 77.3\times$ and `InceptionV3` presented up to $\approx 57.9\times$.

Our observations further support that compiler optimizations can lead to differing outcomes across hardware devices, and that these differences can vary between models. We attribute this behavior to variations in model size, architecture, and computational demands. However, identifying the exact causes of these effects is left to future work, as our focus is to highlight the impact of the computational environment on model correctness and inference time performance.



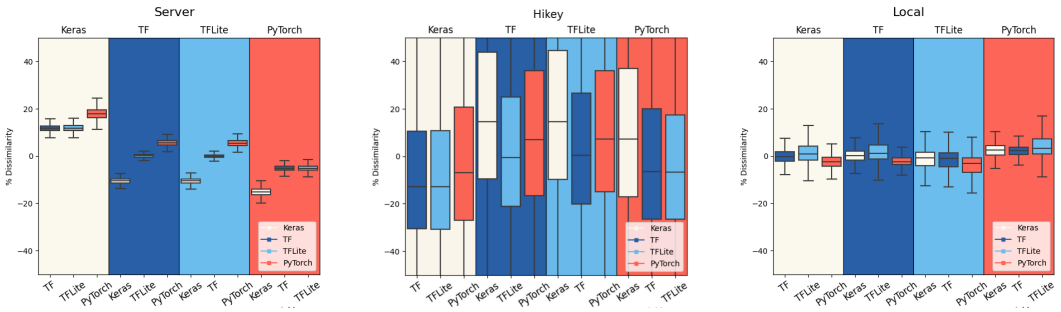
(a) `Server` **(b)** `Hikey` **(c)** `Local`

Figure A.2: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `MobileNetV2`, *Basic* optimization.



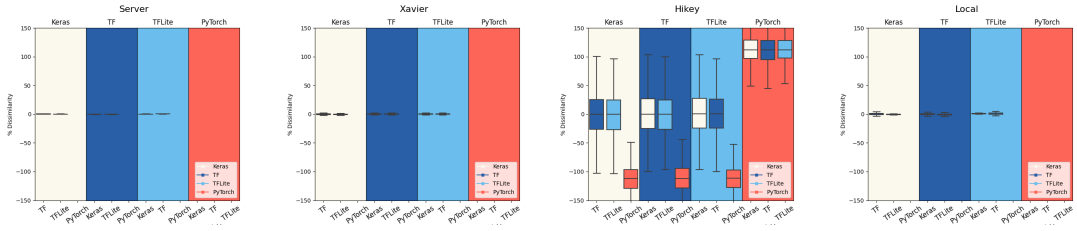
(a) `Server` **(b)** `Hikey` **(c)** `Local`

Figure A.3: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `MobileNetV2`, *Default* optimization.



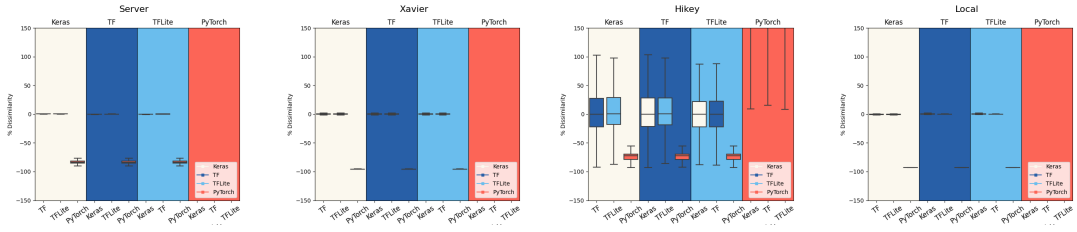
(a) `Server` **(b)** `Hikey` **(c)** `Local`

Figure A.4: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `MobileNetV2`, *Extended* optimization.



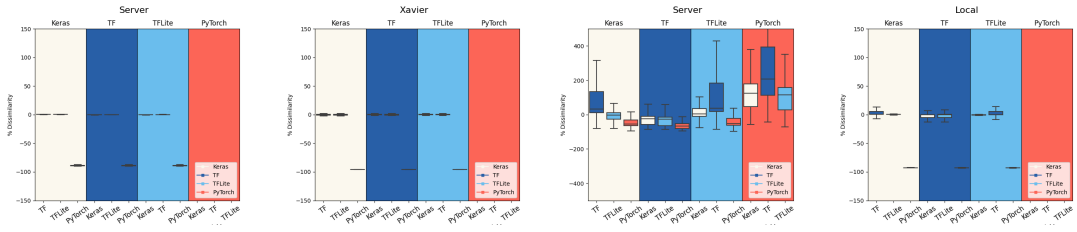
(a) `Server` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.5: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `ResNet101V2`, *Basic* optimization.



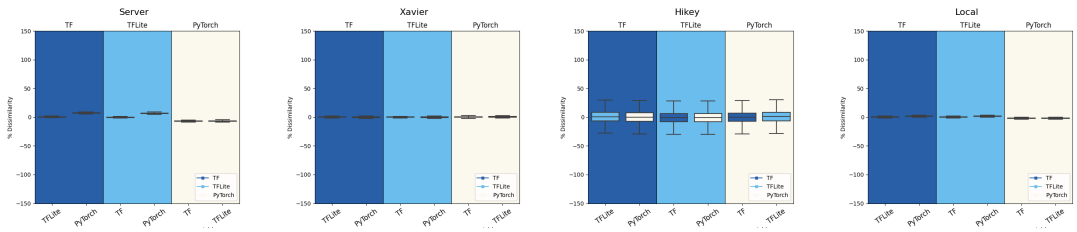
(a) `Server` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.6: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `ResNet101V2`, *Default* optimization.



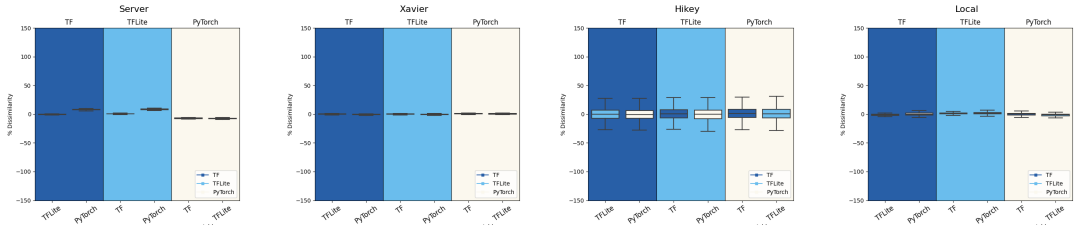
(a) `Server` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.7: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, `ResNet101V2`, *Extended* optimization.



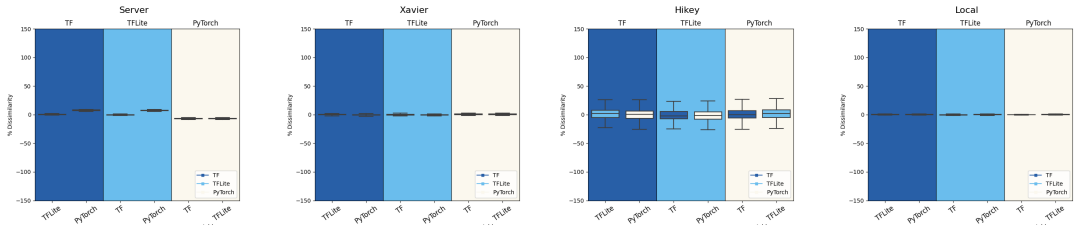
(a) `Server` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.8: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, *InceptionV3*, *Basic* optimization.



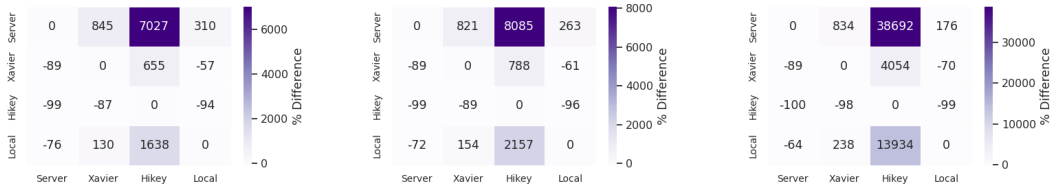
(a) `Server` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.9: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, *InceptionV3*, *Default* optimization.



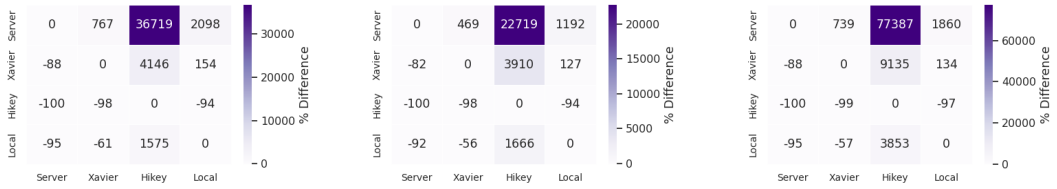
(a) `Sovu` **(b)** `Xavier` **(c)** `Hikey` **(d)** `Local`

Figure A.10: Pairwise comparison of execution times across DNN Frameworks for each hardware acceleration device, *InceptionV3*, *Extended* optimization.



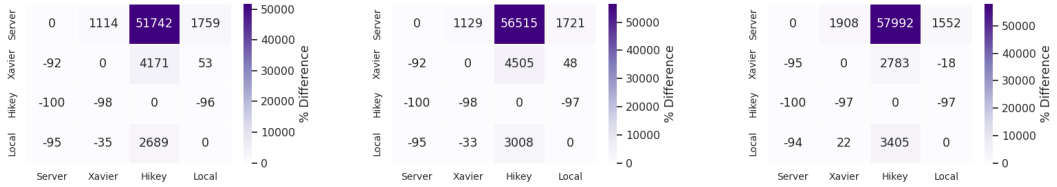
(a) *Basic* optimization **(b)** *Default* optimization **(c)** *Extended* optimization

Figure A.11: Pairwise comparison of execution times across hardware acceleration devices per-optimization setting for `MobileNetV2` (`PyTorch`).



(a) *Basic* optimization **(b)** *Default* optimization **(c)** *Extended* optimization

Figure A.12: Pairwise comparison of execution times across hardware acceleration devices per-optimization setting for `ResNet101V2` (`PyTorch`).



(a) *Basic* optimization **(b)** *Default* optimization **(c)** *Extended* optimization

Figure A.13: Pairwise comparison of execution times across hardware acceleration devices per-optimization setting for *InceptionV3* (`PyTorch`).

Bibliography

[1] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," Insights into Imaging, vol. 9, pp. 611 – 629, 2018.

[2] S. Suhaib Kamran, A. Haleem, S. Bahl, M. Javaid, C. Prakash, and D. Budhhi, "Artificial intelligence and advanced materials in automotive industry: Potential applications and perspectives," Materials Today: Proceedings, vol. 62, pp. 4207–4214, 2022. International Conference on Materials, Processing & Characterization (13th ICMPC).

[3] A. Bohr and K. Memarzadeh, "Chapter 2 - the rise of artificial intelligence in healthcare applications," in Artificial Intelligence in Healthcare (A. Bohr and K. Memarzadeh, eds.), pp. 25–60, Academic Press, 2020.

[4] S. Kumar and R. Tomar, "The Role of Artificial Intelligence In Space Exploration," in 2018 International Conference on Communication, Computing and Internet of Things (IC3IoT), pp. 499–503, 2018.

[5] M. Barenkamp, J. Rebstadt, and O. Thomas, "Applications of AI in classical software engineering," AI Perspectives, vol. 2, 2020.

[6] U. Ergin, "One of the First Fatalities of a Self-Driving Car: Root Cause Analysis of the 2016 Tesla Model S 70D Crash," Trafik ve Ulaşım Araştırmaları Dergisi, 2022.

[7] W. Zhao, S. Alwidian, and Q. Mahmoud, "Adversarial training methods for deep learning: A systematic review," Algorithms, vol. 15, p. 283, 08 2022.

[8] T. K. Dang, P. T. T. Truong, and P. T. Tran, "Data poisoning attack on deep neural network and some defense methods," in 2020 International Conference on Advanced Computing and Applications (ACOMP), pp. 15–22, 2020.

[9] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 2013.

[10] F. Marulli, L. Verde, and L. Campanile, "Exploring Data and Model Poisoning Attacks to Deep Learning-Based NLP Systems," Procedia Computer Science, vol. 192, pp. 3570–3579, 2021. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 25th International Conference KES2021.

[11] N. Humbatova, G. Jahangirova, and P. Tonella, "DeepCrime: mutation testing of deep learning systems based on real faults," in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021, (New York, NY, USA), p. 67–78, Association for Computing Machinery, 2021.

[12] N. Humbatova, G. Jahangirova, and P. 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), pp. 68–72, 2023.

[13] S. Gu, Z. Zong, F. Tian, and Z. Chen, "Dvtest: Deep neural network visualization testing framework," in 2023 10th International Conference on Dependable Systems and Their Applications (DSA), pp. 1–11, 2023.

[14] S. Pavlitska, N. Lambing, and J. M. Zöllner, "Adversarial attacks on traffic sign recognition: A survey," 2023.

[15] F. Khalid, M. A. Hanif, and M. Shafique, "Exploiting Vulnerabilities in Deep Neural Networks: Adversarial and Fault-Injection Attacks," 2021.

[16] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 132–142, IEEE, 2018.

[17] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1027–1038, 2019.

[18] Z. Wang, M. Yan, J. Chen, S. Liu, and D. 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, ESEC/FSE 2020, (New York, NY, USA), p. 788–799, Association for Computing Machinery, 2020.

[19] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," Proceedings of the ACM on measurement and analysis of computing systems, 2022.

[20] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing Deep Learning Compilers with HirGen," International Symposium on Software Testing and Analysis, 2023.

[21] W. Li, Y. Wang, K. Zou, H. Li, and X. Li, "Adversarial testing: A novel on-line testing method for deep learning processors," in 2023 IEEE 32nd Asian Test Symposium (ATS), pp. 1–6, 2023.

[22] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Testing for Multiple Faults in Deep Neural Networks," IEEE Design & Test, vol. 41, no. 3, pp. 47–53, 2024.

[23] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, "Software testing techniques: A literature review," in 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), pp. 177–182, 2016.

[24] "Ariane 501 - Presentation of Inquiry Board Report." https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report, 1996. [Accessed 7-Oct-2024].

[25] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," Computer, vol. 26, no. 7, pp. 18–41, 1993.

[26] Y. L. Karpov, Y. L. Karpov, L. E. Karpov, L. E. Karpov, Y. G. Smetanin, and Y. G. Smetanin, "Adaptation of General Concepts of Software Testing to Neural Networks," Programming and Computer Software, 2018.

[27] D. Marijan, A. Gotlieb, and M. Kumar Ahuja, "Challenges of Testing Machine Learning Based Systems," in 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), pp. 101–102, 2019.

[28] J. Sekhon, J. Sekhon, C. Fleming, and C. Fleming, "Towards improved testing for deep learning," null, 2019.

[29] P. Koopman, P. Koopman, M. Wagner, and M. Wagner, "Challenges in Autonomous Vehicle Testing and Validation," SAE International journal of transportation safety, 2016.

[30] T. Mihalj, D. Nalic, S. Arefnezhad, and A. Eichberger, "Hazards Identification Using Scenario-Based Testing with Respect to ISO PAS 21448 and ISO 26262," in 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC), pp. 5764–5770, 2023.

[31] International Organization for Standardization, "ISO 26262:2018 — Road vehicles — Functional safety." https://www.iso.org/standard/68383.html, 2018. Accessed: 2025-05-13.

[32] Q. V. E. Hommes, "Review and Assessment of the ISO 26262 Draft Road Vehicle - Functional Safety," null, 2012.

[33] International Organization for Standardization, "ISO 21448:2022 — Road vehicles — Safety of the intended functionality." https://www.iso.org/standard/77490.html, 2022. Accessed: 2025-05-13.

[34] L. Birkemeyer, C. King, and I. Schaefer, "Is Scenario Generation Ready for SOTIF? A Systematic Literature Review," in 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC), pp. 472–479, 2023.

[35] S. of Automotive Engineers (SAE), "SAE J3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems." https://www.sae.org/standards/content/j3016_201806/, 2018. Accessed: 2025-05-13.

[36] International Organization for Standardization, "ISO/IEC TR 24029-2:2023 — Artificial Intelligence (AI) — Assessment of the robustness of neural networks — Part 2: Methodology for the use of formal methods." https://www.iso.org/standard/79804.html, 2023. Accessed: 2025-05-13.

[37] P. Koopman, "UL 4600: What to Include in an Autonomous Vehicle Safety Case," Computer, vol. 56, no. 5, pp. 101–104, 2023.

[38] J. Athavale and D. Galpin, " Functional Safety Standards: IEEE P2851 Road Map ," Computer, vol. 58, pp. 158–160, Apr. 2025.

[39] P. Buddi, C. V. K. N. S. N. Moorthy, N. Venkateswarulu, and D. Myneni, "Exploration of issues, challenges and latest developments in autonomous cars," Journal of Big Data, vol. 10, 05 2023.

[40] L. Ma, F. Juefei-Xu, M. Xue, Q. Hu, S. Chen, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Secure deep learning engineering: A software quality assurance perspective," 2018.

[41] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1110–1121, 2020.

[42] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A Comprehensive Study on Challenges in Deploying Deep Learning Based Software," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 750–762, 2020.

[43] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," 2019.

[44] X. Wei, Y. Guo, and J. Yu, "Adversarial Sticker: A Stealthy Attack Method in the Physical World," 2022.

[45] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," CoRR, vol. abs/1705.06640, 2017.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," CoRR, vol. abs/1912.01703, 2019.

[47] F. e. a. Chollet, "Keras." `https://keras.io`, 2015.

[48] L. Bouzar-Benlabiod, S. H. Rubin, and A. Benaida, "Optimizing Deep Neural Network Architectures: an overview," in 2021 IEEE 22nd International Conference on Information Reuse and Integration for Data Science (IRI), pp. 25–32, 2021.

[49] "OpenCL." `https://www.khronos.org/opencl/`. Accessed 7 Oct. 2024.

[50] "CUDA." `https://developer.nvidia.com/cuda-toolkit`. Accessed 7 Oct. 2024.

[51] Martín Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. Software available from tensorflow.org.

[52] "Converting tensorflow model to pytorch: issue with padding." https://discuss.pytorch.org/t/converting-tensorflow-model-to-pytorch-issue-with-padding/84224, 2020. [Accessed 7-Oct-2024].

[53] Y. Liu, C. Chen, R. Zhang, T. Qin, X. Ji, H. Lin, and M. Yang, "Enhancing the interoperability between deep learning frameworks by model conversion," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, (New York, NY, USA), p. 1320–1330, Association for Computing Machinery, 2020.

[54] "TF2ONNX." https://github.com/onnx/tensorflow-onnx, 2022. [Accessed 15-Feb-2023].

[55] "tflite2onnx." https://github.com/zhenhuaw-me/tflite2onnx, 2020. [Accessed 6-June-2023].

[56] "onnx2keras." https://github.com/gmalivenko/onnx2keras, 2023. [Accessed 15-Feb-2023].

[57] "onnx2torch." https://github.com/ENOT-AutoDL/onnx2torch, 2023. [Accessed 15-Feb-2023].

[58] "Open Neural Network Exchange." https://onnx.ai/, 2023. [Accessed 8-Dec-2023].

[59] IEEE, "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84, 2019.

[60] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 578–594, Oct. 2018.

[61] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," 2020.

[62] A. F. Donaldson, "Metamorphic testing of android graphics drivers," in 2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET), pp. 1–1, 2019.

[63] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," Proc. ACM Program. Lang., vol. 1, Oct. 2017.

[64] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," SIGPLAN Not., vol. 50, p. 65–76, June 2015.

[65] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," SIGPLAN Not., vol. 46, p. 283–294, June 2011.

[66] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," CoRR, vol. abs/1512.03385, 2015.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," CoRR, vol. abs/1603.05027, 2016.

[68] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-Scale Hierarchical Image Database," in 2009 IEEE conference on computer vision and pattern recognition, pp. 248–255, 2009.

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

[70] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "DeltaNN: Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models," in 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), (Los Alamitos, CA, USA), pp. 414–424, IEEE Computer Society, Oct. 2023.

[71] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," CoRR, vol. abs/1801.04381, 2018.

[72] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," CoRR, vol. abs/1512.00567, 2015.

[73] "OpenAI Triton." `https://openai.com/index/triton/`. Accessed 9 Oct 2024.

[74] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Assessing Robustness of Image Recognition Models to Changes in the Computational Environment," in NeurIPS ML Safety Workshop, 2022.

[75] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Exploring Effects of Computational Parameter Changes to Image Recognition Systems," arXiv preprint arXiv:2211.00471, 2022.

[76] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1795–1799, 2023.

[77] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "FetaFix: Automatic Fault Localization and Repair of Deep Learning Model Conversions," 2025.

[78] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Exploring Robustness of Image Recognition Models on Hardware Accelerators," in 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 368–374, 2025.

[79] P. Gibson, J. Cano, E. J. Crowley, A. Storkey, and M. O'Boyle, "DLAS: A Conceptual Model for Across-Stack Deep Learning Acceleration," ACM Transactions on Architecture and Code Optimization (TACO), 2025.

[80] Y. Sun, X. Huang, and D. Kroening, "Testing Deep Neural Networks," arXiv: Learning, 2018.

[81] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," 2022.

[82] A. Arrieta, P. Valle, A. Iriarte, and M. Illarramendi, "How Do Deep Learning Faults Affect AI-Enabled Cyber-Physical Systems in Operation? A Preliminary Study Based on DeepCrime Mutation Operators," in 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–7, 2023.

[83] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 674–685, IEEE, 2021.

[84] B. D. Evans, G. Malhotra, and J. S. Bowers, "Biological convolutions improve DNN robustness to noise and generalisation," Neural Networks, vol. 148, pp. 96–110, 2022.

[85] J. Guo, Y. Zhao, H. Song, and Y. Jiang, "Coverage Guided Differential Adversarial Testing of Deep Learning Systems," IEEE Transactions on Network Science and Engineering, vol. 8, pp. 933–942, Apr. 2021.

[86] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepMutation: Mutation Testing of Deep Learning Systems," in 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), (Los Alamitos, CA, USA), pp. 100–111, IEEE Computer Society, oct 2018.

[87] H. F. Eniser, S. Gerasimou, and A. Sen, "DeepFault: Fault Localization for Deep Neural Networks," in Fundamental Approaches to Software Engineering, pp. 171–191, 2019.

[88] M. Wardat, W. Le, and H. Rajan, "DeepLocalize: Fault Localization for Deep Neural Networks," in Proceedings of the 43rd International Conference on Software Engineering, p. 251–262, 2021.

[89] F. Jafarinejad, K. Narasimhan, and M. Mezini, "NerdBug: automated bug detection in neural networks," in Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis, AISTA 2021, (New York, NY, USA), p. 13–16, Association for Computing Machinery, 2021.

[90] A. Ghanbari, D.-G. Thomas, M. A. Arshad, and H. Rajan, "Mutation-based Fault Localization of Deep Neural Networks," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1301–1313, 2023.

[91] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization," in 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 153–162, 2014.

[92] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," Softw. Test. Verif. Reliab., vol. 25, p. 605–628, Aug. 2015.

[93] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: precision testing of deep learning operators," in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021, (New York, NY, USA), p. 400–412, Association for Computing Machinery, 2021.

[94] J. Chen, C. Jia, Y. Yan, J. Ge, H. Zheng, and Y. Cheng, "A Miss Is as Good as A Mile: Metamorphic Testing for Deep Learning Operators," Proc. ACM Softw. Eng., vol. 1, July 2024.

[95] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1174–1186, 2023.

[96] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, pp. 533–536, 1986.

[97] H. Wu, Z. Li, Z. Cui, and J. Zhang, "A Mutation-based Approach to Repair Deep Neural Network Models," in 2021 8th International Conference on Dependable Systems and Their Applications (DSA), pp. 730–731, 2021.

[98] J. Sohn, S. Kang, and S. Yoo, "Arachne: Search Based Repair of Deep Neural Networks," ACM Transactions on Software Engineering and Methodology, 2022.

[99] D. Li Calsi, T. Laurent, P. Arcaini, and F. Ishikawa, "Federated Repair of Deep Neural Networks," in Proceedings of the 5th IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning, DeepTest '24, (New York, NY, USA), p. 17–24, Association for Computing Machinery, 2024.

[100] M. Wardat and A. Al-Alaj, "DeepCNN: A Dual Approach to Fault Localization and Repair in Convolutional Neural Networks," IEEE Access, vol. 12, pp. 50321–50334, 2024.

[101] J. Ma, P. Yang, J. Wang, Y. Sun, C.-C. Huang, and Z. Wang, "VeRe: Verification Guided Synthesis for Repairing Deep Neural Networks," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[102] D. Li Calsi, M. Duran, T. Laurent, X.-Y. Zhang, P. Arcaini, and F. Ishikawa, "Adaptive Search-based Repair of Deep Neural Networks," in Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '23, (New York, NY, USA), p. 1527–1536, Association for Computing Machinery, 2023.

[103] J. Kim, N. Humbatova, G. Jahangirova, P. Tonella, and S. Yoo, "Repairing DNN Architecture: Are We There Yet?," in 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 234–245, 2023.

[104] W. M. McKeeman and W. M. McKeeman, "Differential Testing for Software," Digital Technical Journal, 1998.

[105] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in Proceedings of the 40th international conference on software engineering, pp. 303–314, 2018.

[106] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, "EAGLE: creating equivalent graphs to test deep learning libraries," in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, (New York, NY, USA), p. 798–810, Association for Computing Machinery, 2022.

[107] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: testing deep learning libraries via neural architecture fuzzing," in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, (New York, NY, USA), p. 1418–1430, Association for Computing Machinery, 2022.

[108] A. Prochnow and J. Yang, "Diffwatch: watch out for the evolving differential testing in deep learning libraries," in Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, ICSE '22, (New York, NY, USA), p. 46–50, Association for Computing Machinery, 2022.

[109] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019 (S. Kraus, ed.), IJCAI International Joint Conference on Artificial Intelligence, pp. 5772–5778, International Joint Conferences on Artificial Intelligence, 2019.

[110] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," Commun. ACM, vol. 33, no. 12, pp. 32–44, 1990.

[111] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, 2011.

[112] N. Chetouane, L. Klampfl, and F. Wotawa, "Investigating the Effectiveness of Mutation Testing Tools in the Context of Deep Neural Networks," in Advances in Computational Intelligence (I. Rojas, G. Joya, and A. Catala, eds.), (Cham), pp. 766–777, Springer International Publishing, 2019.

[113] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "DeepMutation++: A Mutation Testing Framework for Deep Learning Systems," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1158–1161, 2019.

[114] W. Shen, J. Wan, and Z. Chen, "MuNN: Mutation Analysis of Neural Networks," in 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 108–115, 2018.

[115] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," null, 2018.

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

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

[118] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," 2020.

[119] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in Advances in Neural Information Processing Systems (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[120] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing Deep Neural Networks: Fix Patterns and Challenges," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, (New York, NY, USA), p. 1135–1146, Association for Computing Machinery, 2020.

[121] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A Comprehensive Study on Deep Learning Bug Characteristics," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, (New York, NY, USA), p. 510–520, Association for Computing Machinery, 2019.

[122] N. Papernot, P. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical Black-Box Attacks against Machine Learning," null, 2017.

[123] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," null, 2017.

[124] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: systematic physical-world testing of autonomous driving systems," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, (New York, NY, USA), p. 347–358, Association for Computing Machinery, 2020.

[125] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," 2018.

[126] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," 2018.

[127] H. Ben Braiek and F. Khomh, "DeepEvolution: A Search-Based Testing Approach for Deep Neural Networks," in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 454–458, 2019.

[128] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "RobOT: Robustness-Oriented Testing for Deep Learning Systems," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 300–311, 2021.

[129] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: a simple and accurate method to fool deep neural networks," 2016.

[130] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "DeepStellar: model-based quantitative analysis of stateful deep learning systems," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, (New York, NY, USA), p. 477–487, Association for Computing Machinery, 2019.

[131] X. Du, X. Xie, Y. Li, L. Ma, J. Zhao, and Y. Liu, "DeepCruiser: Automated Guided Testing for Stateful Deep Learning Systems," 2018.

[132] C. Shorten and T. M. Khoshgoftaar, "A Survey on Image Data Augmentation for Deep Learning," Journal of big data, vol. 6, no. 1, pp. 1–48, 2019.

[133] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepGauge: multi-granularity testing criteria for deep learning systems," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, (New York, NY, USA), p. 120–131, Association for Computing Machinery, 2018.

[134] G. Han, Z. Li, P. Tang, C. Hu, and S. Guo, "FuzzGAN: A Generation-Based Fuzzing Framework for Testing Deep Neural Networks," in 2022 IEEE 24th International Conference 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), pp. 1601–1608, 2022.

[135] A. Odena and I. Goodfellow, "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing," 2018.

[136] J. Yu, Y. Fu, Y. Zheng, Z. Wang, and X. Ye, "Test4Deep: an Effective White-Box Testing for Deep Neural Networks," in 2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), pp. 16–23, 2019.

[137] J. Yu, J. Yu, S. Duan, S. Duan, X. Ye, and X. Yao, "A White-Box Testing for Deep Neural Networks Based on Neuron Coverage.," IEEE transactions on neural networks and learning systems, 2022.

[138] D. Wang, D. Wang, Z. Wang, W. Ziyuan, Z. Wang, C. Fang, C. Fang, C. Fang, Y. Chen, Y. Chen, Z. Chen, and Z. Chen, "DeepPath: Path-Driven Testing Criteria for Deep Neural Networks," International Conference on Artificial Intelligence Testing, 2019.

[139] S. Lee, S. Cha, D. Lee, and H. Oh, "Effective white-box testing of deep neural networks with adaptive neuron-selection strategy," in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, (New York, NY, USA), p. 165–176, Association for Computing Machinery, 2020.

[140] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks," in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, (New York, NY, USA), p. 177–188, Association for Computing Machinery, 2020.

[141] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "DeepHunter: a coverage-guided fuzz testing framework for deep neural networks," in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, (New York, NY, USA), p. 146–157, Association for Computing Machinery, 2019.

[142] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "DeepConcolic: Testing and Debugging Deep Neural Networks," in 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 111–114, 2019.

[143] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, (New York, NY, USA), p. 109–119, Association for Computing Machinery, 2018.

[144] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems," in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 614–618, 2019.

[145] H. Deokuliar, R. S. Sangwan, Y. Badr, and S. M. Srinivasan, "Improving Testing of Deep-learning Systems: A combination of differential and mutation testing results in better test data.," Queue, vol. 21, p. 54–65, nov 2023.

[146] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, May 2019.

[147] Y. Lu, K. Shao, W. Sun, and M. Sun, "RGChaser: A RL-guided Fuzz and Mutation Testing Framework for Deep Learning Systems," in 2022 9th International Conference on Dependable Systems and Their Applications (DSA), pp. 12–23, 2022.

[148] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," in 2017 IEEE International Conference on Computer Vision (ICCV), pp. 618–626, 2017.

[149] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray, "Testing DNN image classifiers for confusion & bias errors," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, (New York, NY, USA), p. 1122–1134, Association for Computing Machinery, 2020.

[150] T. Sellam, K. Lin, I. Y. Huang, Y. Chen, M. Yang, C. Vondrick, and E. Wu, "DeepBase: Deep Inspection of Neural Networks," 2019.

[151] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "MODE: automated neural network model debugging via state differential analysis and input selection," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, (New York, NY, USA), p. 175–186, Association for Computing Machinery, 2018.

[152] Y. Sun, H. Chockler, X. Huang, and D. Kroening, "Explaining Image Classifiers using Statistical Fault Localization," 2020.

[153] D. Gopinath, M. Zhang, K. Wang, I. B. Kadron, C. Pasareanu, and S. Khurshid, "Symbolic Execution for Importance Analysis and Adversarial Generation in Neural Networks," in 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 313–322, 2019.

[154] Y. Qin, H. Wang, C. Xu, X. Ma, and J. Lu, "Syneva: Evaluating ml programs by mirror program synthesis," null, 2018.

[155] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 810–822, 2019.

[156] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," CoRR, vol. abs/1608.07249, 2016.

[157] L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin, and Q. Zhang, "Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond," in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1258–1269, 2018.

[158] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. F. O'Boyle, "M3: Semantic API Migrations," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 90–102, 2020.

[159] P. Stratis and A. Rajan, "Speeding Up Test Execution with Increased Cache Locality," Software Testing, Verification and Reliability, vol. 28, no. 5, p. e1671, 2018.

[160] N. Mahmoud, Y. Essam, R. Elshawi, and S. Sakr, "DLBench: An Experimental Evaluation of Deep Learning Frameworks," in 2019 IEEE International Congress on Big Data, pp. 149–156, 2019.

[161] Y. Wu, L. Liu, C. Pu, W. Cao, S. Sahin, W. Wei, and Q. Zhang, "A Comparative Measurement Study of Deep Learning as a Service Framework," IEEE Transactions on Services Computing, vol. 15, no. 1, pp. 551–566, 2022.

[162] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, (New York, NY, USA), p. 129–140, Association for Computing Machinery, 2018.

[163] M. Nejadgholi and J. Yang, "A Study of Oracle Approximations in Testing Deep Learning Libraries," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 785–796, 2019.

[164] S.-S. Kim, S. Kim, H. Wimmer, H. Wimmer, J. Kim, and J. Kim, "Analysis of Deep Learning Libraries: Keras, PyTorch, and MXnet," International Conference on Software Engineering Research and Applications, 2022.

[165] F. Florencio, T. Silva, E. Ordonez, and M. Júnior, "Performance Analysis of Deep Learning Libraries: TensorFlow and PyTorch," Journal of Computer Science, vol. 15, 05 2019.

[166] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward Understanding Deep Learning Framework Bugs," ACM Transactions on Software Engineering and Methodology, 2023.

[167] X. Shen, J. Zhang, X. Wang, H. Yu, and G. Sun, "Deep Learning Framework Fuzzing Based on Model Mutation," in 2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC), pp. 375–380, 2021.

[168] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated Testing for Deep Learning Frameworks," in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 486–498, 2020.

[169] J. Li, S. Li, J. Wu, L. Luo, Y. Bai, and H. Yu, "MMOS: Multi-Staged Mutation Operator Scheduling for Deep Learning Library Testing," in GLOBECOM 2022 - 2022 IEEE Global Communications Conference, pp. 6103–6108, 2022.

[170] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "COMET: Coverage-guided Model Generation For Deep Learning Library Testing," ACM Trans. Softw. Eng. Methodol., vol. 32, July 2023.

[171] W. Luo, D. Chai, X. Ruan, J. Wang, C. Fang, and Z. Chen, "Graph-Based Fuzz Testing for Deep Learning Inference Engines," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 288–299, 2021.

[172] S. Chorev, P. Tannor, D. B. Israel, N. Bressler, I. Gabbay, N. Hutnik, J. Liberman, M. Perlmutter, Y. Romanyshyn, and L. Rokach, "Deepchecks: A Library for Testing and Validating Machine Learning Models and Data," 2022.

[173] E. Schoop, F. Huang, and B. Hartmann, "UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior," in Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[174] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "DeepDiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs," in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, (New York, NY, USA), p. 561–572, Association for Computing Machinery, 2022.

[175] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "DeepFD: automated fault diagnosis and localization for deep learning programs," in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, (New York, NY, USA), p. 573–585, Association for Computing Machinery, 2022.

[176] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, "Exposing numerical bugs in deep learning via gradient back-propagation," in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, (New York, NY, USA), p. 627–638, Association for Computing Machinery, 2021.

[177] J. Shi, Y. Xiao, Y. Li, Y. Li, D. Yu, C. Yu, H. Su, Y. Chen, and W. Huo, "ACETest: Automated Constraint Extraction for Testing Deep Learning Operators," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, (New York, NY, USA), p. 690–702, Association for Computing Machinery, 2023.

[178] J. Wu, S. Li, J. Li, L. Luo, H. Yu, and G. Sun, "DeepCov: Coverage Guided Deep Learning Framework Fuzzing," in 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC), pp. 399–404, 2022.

[179] J. Liu, Y. Huang, Z. Wang, L. Ma, C. Fang, M. Gu, X. Zhang, and Z. Chen, "Generation-based Differential Fuzzing for Deep Learning Libraries," ACM Trans. Softw. Eng. Methodol., vol. 33, Dec. 2023.

[180] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, (New York, NY, USA), p. 423–435, Association for Computing Machinery, 2023.

[181] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: fuzzing deep-learning libraries from open source," in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, (New York, NY, USA), p. 995–1007, Association for Computing Machinery, 2022.

[182] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational API inference," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, (New York, NY, USA), p. 44–56, Association for Computing Machinery, 2022.

[183] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "DocTer: documentation-guided fuzzing for testing deep learning API functions," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, (New York, NY, USA), p. 176–188, Association for Computing Machinery, 2022.

[184] D. Xie, J. Wang, H. V. Pham, L. Tan, Y. Guo, A. Aziz, and E. Meijer, "CEDAR: Continuous Testing of Deep Learning Libraries," in 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 371–382, 2024.

[185] L.-C. Feng, X.-Y. Wang, S.-Y. Zhang, R.-Z. Gao, and Z.-H. Zhao, "Mutation Operator Reduction for Cost-effective Deep Learning Software Testing via Decision Boundary Change Measurement," Journal of Internet Technology, 2022.

[186] "Torch." https://github.com/torch/torch7, 2012. [Accessed 23-Nov-2024].

[187] "Loading a TensorFlow checkpoint, and turn it into a Keras model." https://github.com/keras-team/keras/issues/5273, 2022. [Accessed 24-Aug-2024].

[188] M. Openja, A. Nikanjam, A. H. Yahmed, F. Khomh, and Z. M. J. Jiang, "An Empirical Study of Challenges in Converting Deep Learning Models," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 13–23, 2022.

[189] P. Jajal, W. Jiang, A. Tewari, E. Kocinare, J. Woo, A. Sarraf, Y.-H. Lu, G. K. Thiruvathukal, and J. C. Davis, "Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters," in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, (New York, NY, USA), p. 1466–1478, Association for Computing Machinery, 2024.

[190] Q. Shen et al., "A comprehensive study of deep learning compiler bugs," in 29th ACM ESEC/FSE, ESEC/FSE 2021, (New York, NY, USA), p. 968–980, ACM, 2021.

[191] "ONNX Optimizer." `https://github.com/onnx/optimizer`, 2020. [Accessed 23-Apr-2025].

[192] "NVIDIA." https://www.nvidia.com/, 2024. [Accessed 03-Jul-2023].

[193] "AMD." https://www.amd.com/, 2024. [Accessed 03-Jul-2023].

[194] P. Omland, Y. Peng, M. Paulitsch, J. Parra, G. Espinosa, A. Daniel, G. Hinz, and A. Knoll, "API-Based Hardware Fault Simulation for DNN Accelerators," IEEE Design & Test, vol. 40, no. 2, pp. 75–81, 2023.

[195] M. H. Rahman, S. Laskar, and G. Li, "Investigating the impact of transient hardware faults on deep learning neural network inference," Software testing, verification & reliability, 2024.

[196] A. Chaudhuri, J. Talukdar, and K. Chakrabarty, "Machine Learning for Testing Machine-Learning Hardware: A Virtuous Cycle," in Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[197] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Compact Test Pattern Generation For Multiple Faults In Deep Neural Networks," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–2, 2023.

[198] Z. Wang, P. Nie, X. Miao, Y. Chen, C. Wan, L. Bu, and J. Zhao, "GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, (New York, NY, USA), p. 904–916, Association for Computing Machinery, 2023.

[199] "TVMFuzz." `https://github.com/dpankratz/TVMFuzz`. Accessed 28 Jan. 2024.

[200] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '23, ACM, Jan. 2023.

[201] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, (New York, NY, USA), p. 95–105, Association for Computing Machinery, 2018.

[202] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, "MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1555–1566, 2023.

[203] C. Suo, J. Chen, S. Liu, J. Jiang, Y. Zhao, and J. Wang, "Fuzzing MLIR Compiler Infrastructure via Operation Dependency Analysis," in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, (New York, NY, USA), p. 1287–1299, Association for Computing Machinery, 2024.

[204] B. Limpanukorn, J. Wang, H. J. Kang, Z. Zhou, and M. Kim, "Fuzzing MLIR Compilers with Custom Mutation Synthesis ," in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), (Los Alamitos, CA, USA), pp. 457–468, IEEE Computer Society, May 2025.

[205] K. Lin, X. Song, Y. Zeng, and S. Guo, "DeepDiffer: Find Deep Learning Compiler Bugs via Priority-guided Differential Fuzzing," in 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), pp. 616–627, 2023.

[206] Q. Su, C. Geng, G. Pekhimenko, and X. Si, "TorchProbe: Fuzzing Dynamic Deep Learning Compilers," Asian Symposium on Programming Languages and Systems, 2023.

[207] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An Empirical Study of Common Challenges in Developing Deep Learning Applications," in 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 104–115, 2019.

[208] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "Mobilenet-SSDv2: An Improved Object Detection Model for Embedded Systems," in 2020 International Conference on System Science and Engineering (ICSSE), pp. 1–5, 2020.

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

[210] Z. Dai, H. Liu, Q. V. Le, and M. Tan, "CoAtNet: Marrying Convolution and Attention for All Data Sizes," in Advances in Neural Information Processing Systems, vol. 34, pp. 3965–3977, 2021.

[211] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling Vision Transformers," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12104–12113, 2022.

[212] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, (Red Hook, NY, USA), p. 3393–3404, Curran Associates Inc., 2018.

[213] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 863–879, 2020.

[214] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient Primitives for Deep Learning," arXiv preprint arXiv:1410.0759, 2014.

[215] "Compute Library." Arm Software, Aug. 2022.

[216] H. Tabani, A. Balasubramaniam, E. Arani, and B. Zonooz, "Challenges and obstacles towards deploying deep learning models on mobile devices," ArXiv, vol. abs/2105.02613, 2021.

[217] M. A. Merzoug, A. Mostefaoui, M. H. Kechout, and S. Tamraoui, "Deep learning for resource-limited devices," in Proceedings of the 16th ACM Symposium on QoS and Security for Wireless and Mobile Networks, Q2SWinet '20, (New York, NY, USA), p. 81–87, Association for Computing Machinery, 2020.

[218] V. Yaneva, A. Rajan, and C. Dubach, "Compiler-Assisted Test Acceleration on GPUs for Embedded Software," in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 35–45, 2017.

[219] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VERIFAI: A Toolkit for the Design and Analysis of Artificial Intelligence-Based Systems," 2019.

[220] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," CoRR, vol. abs/1704.04861, 2017.

[221] C. Szegedy et al., "Going deeper with convolutions," CoRR, vol. abs/1409.4842, 2014.

[222] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A High-Level Compiler for Deep Learning," 2019.

[223] P. Gibson and J. Cano, "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation," in Proceedings of the International Conference on Parallel Architectures and CCompilation Techniques (PACT), p. 28–39, 2023.

[224] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning," Sustainable Computing: Informatics and Systems, vol. 38, p. 100857, Apr. 2023.

[225] C. Warren, A. Giannopoulos, A. Gray, I. Giannakis, A. Patterson, L. Wetter, and A. Hamrah, "A CUDA-based GPU engine for gprMax: Open source FDTD electromagnetic simulation software," Computer Physics Communications, vol. 237, pp. 208–218, 2019.

[226] "NVIDIA Jetson Modules - Technical Specification." https://developer.nvidia.com/embedded/jetson-modules, 2025. [Accessed 14-May-2025].

[227] "The Compute Architecture of Intel Processor Graphics Gen9." https://www.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf, 2025. [Accessed 14-May-2025].

[228] S. Wang, A. Pathania, and T. Mitra, "Neural Network Inference on Mobile SoCs," IEEE Design & Test, vol. 37, p. 50–57, Oct. 2020.

[229] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in 2011 International Conference on Parallel Processing, pp. 216–225, 2011.

[230] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Commun. ACM, vol. 60, p. 84–90, may 2017.

[231] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, " Going deeper with convolutions ," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (Los Alamitos, CA, USA), pp. 1–9, IEEE Computer Society, June 2015.

[232] O. R. developers, "ONNX Runtime." `https://onnxruntime.ai/`, 2021. [Accessed 15-Feb-2023].

[233] "Convert TensorFlow Models." https://www.tensorflow.org/lite/models /convert, 2023. [Accessed 26-June-2023].

[234] J. Camacho-Collados and M. T. Pilehvar, "On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis," CoRR, vol. abs/1707.01780, 2017.

[235] "TensorFlow - Frequently Asked Questions," 2021.

[236] "PyTorch to Keras code equivalence." `https://stackoverflow.com/questions/46866763/pytorch-to-keras-code-equivalence`. Accessed 23 Feb. 2024.

[237] "Convert Keras model to PyTorch." `https://stackoverflow.com/questions/68413480/convert-keras-model-to-pytorch`. Accessed 23 Feb. 2024.

[238] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 183–200, 2002.

[239] P. McMinn, "Search-based software test data generation: a survey: Research articles," Softw. Test. Verif. Reliab., vol. 14, p. 105–156, June 2004.

[240] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 556–559, 2011.

[241] "Always Getting 0 for Prediction from Tensorflow Lite Model." `https://stackoverflow.com/questions/67069167/always-getting-0-for-prediction-from-tensorflow-lite-model`. Accessed 13 Dec. 2023.

[242] "Incorrect Data Response in Tensorflow." https://stackoverflow.com/questions/76418614/incorrect-data-response-in-tensorflow. Accessed 13 Dec. 2023.

[243] "CoreML model converted from PyTorch model giving the wrong prediction probability." https://stackoverflow.com/questions/64519191/coreml-model-converted-from-pytorch-model-giving-the-wrong-prediction-probabilit. Accessed 13 Dec. 2023.

[244] "Want to Confirm if This Is a Problem with Model or I Am Doing Something Wrong (TF)." https://stackoverflow.com/questions/73431543/want-to-confirm-if-this-is-a-problem-with-model-or-i-am-doing-something-wrong-tf. Accessed 13 Dec. 2023.

[245] "Convert to CoreML but Predict Wrong." https://discuss.pytorch.org/t/convert-to-coreml-but-predict-wrong/66355/3. Accessed 13 Dec. 2023.

[246] "TensorFlow Lite model gives very different accuracy value compared to python model." https://stackoverflow.com/questions/52057552/tensorflow-lite-model-gives-very-different-accuracy-value-compared-to-python-mod. Accessed 23 Mar. 2024.

[247] "Cannot set tensor: Dimension mismatch." https://stackoverflow.com/questions/72516622/cannot-set-tensor-dimension-mismatch. Accessed 13 Dec. 2023.

[248] "ValueError: Cannot set tensor: Dimension mismatch." https://discuss.tensorflow.org/t/valueerror-cannot-set-tensor-dimension-mismatch/15313. Accessed 13 Dec. 2023.

[249] "ValueError: Cannot set tensor: Dimension mismatch (3 but expected 4)." https://stackoverflow.com/questions/67068742/valueerror-cannot-set-tensor-dimension-mismatch-got-3-but-expected-4-for-inpu. Accessed 13 Dec. 2023.

[250] "TensorFlow/Keras ValueError on Input Shape." https://stackoverflow.com/questions/68837658/tensorflow-keras-valueerror-on-input-shape. Accessed 13 Dec. 2023.

[251] "Dimension mismatch during Keras to ONNX conversion (2D output)." https://stackoverflow.com/questions/70861809/dimension-mismatch-during-keras-to-onnx-conversion-2d-output. Accessed 13 Dec. 2023.

[252] "Keras model's summary, first output shape is [(none, 1, 28, 28)] - issue #104 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/104. Accessed 13 Dec. 2023.

[253] "Model gets correct input dimensions, but throws dimension error." https://stackoverflow.com/questions/56292213/model-gets-correct-input-dimensions-but-throws-dimension-error. Accessed 13 Dec. 2023.

[254] "PyTorch to ONNX to TensorFlow - How to Convert from NCHW (ONNX) to NHWC (TensorFlow Lite) - Issue #862 - Onnx/onnx-tensorflow." https://github.com/onnx/onnx-tensorflow/issues/862. Accessed 13 Dec. 2023.

[255] "Strange dimension behaviour: Needs both dimension 2 and 3, unsure why." https://stackoverflow.com/questions/58031343/strange-dimension-behaviour-needs-both-dimension-2-and-3-unsure-why. Accessed 13 Dec. 2023.

[256] "Custom Converter Being Wrapped by Transpose Statements (set_converter) - Issue #572 - Onnx/keras-onnx." https://github.com/onnx/keras-onnx/issues/572. Accessed 13 Dec. 2023.

[257] "Error: Failing in transpose layer (cannot permute batch dimension. result may be wrong) - issue #31 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/31. Accessed 13 Dec. 2023.

[258] "Layer weight shape don't match - issue #78 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/78. Accessed 13 Dec. 2023.

[259] "Reshape after view is wrong - issue #76 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/76. Accessed 13 Dec. 2023.

[260] "Tensorflow to Caffe, reshape layer - issue #831 - Microsoft/MMdnn." https://github.com/microsoft/MMdnn/issues/831. Accessed 13 Dec. 2023.

[261] "Why Does Onnx-tensorflow Add Transpose Layers for Each Conv2D Layer? - Issue #782 - Onnx/onnx-tensorflow." https://github.com/onnx/onnx-tensorflow/issues/782. Accessed 13 Dec. 2023.

[262] "Converted keras model has different parameters - issue #127 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/127. Accessed 13 Dec. 2023.

[263] "Converted model has different weights than the original model - issue #124 - gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/124. Accessed 13 Dec. 2023.

[264] "Different accuracy after model conversion from Keras to Caffe - issue #823 - Microsoft/MMdnn." https://github.com/microsoft/MMdnn/issues/823. Accessed 13 Dec. 2023.

[265] "Failed to convert weights to 8 bit precision: "quantize weights tool only supports tflite models with one subgraph" - issue #35194 - tensorflow/tensorflow." https://github.com/tensorflow/tensorflow/issues/35194. Accessed 13 Dec. 2023.

[266] "After converting the model to .tflite and running it on Android, the accuracy drops." https://discuss.tensorflow.org/t/after-converting-the-model-to-tflite-and-running-it-on-android-the-accuracy-drops/1310/2. Accessed 13 Dec. 2023.

[267] "TensorFlow Lite conversion changes model weights." https://stackoverflow.com/questions/54404262/tensorflow-lite-conversion-changes-model-weights. Accessed 13 Dec. 2023.

[268] "TFLite Model Overflows on GPU, OK on CPU - What Are the Differences Internally?." https://stackoverflow.com/questions/62032560/tflite-model-overflows-on-gpu-ok-on-cpu-what-are-the-differences-internally. Accessed 13 Dec. 2023.

[269] "Accuracy drop between TensorFlow model and converted TFLite." https://stackoverflow.com/questions/65731362/accuracy-drop-between-tensorflow-model-and-converted-tflite. Accessed 13 Dec. 2023.

[270] "Extreme model accuracy loss due to TFLite conversion w/ quantization." https://discuss.tensorflow.org/t/extreme-model-accuracy-loss-due-to-tflite-conversion-w-quantization/2637/5. Accessed 13 Dec. 2023.

[271] "TFLite Output Different Result with Pbfile when Using Only One Convolutional Layer? - Issue #31359 - Tensorflow/tensorflow." https://github.com/tensorflow/tensorflow/issues/31359. Accessed 13 Dec. 2023.

[272] "TFLite: Changing weights - issue #31205 - tensorflow/tensorflow." https://github.com/tensorflow/tensorflow/issues/31205. Accessed 13 Dec. 2023.

[273] "Weights are not equal when convert model from Tensorflow to Caffe - issue #297 - Microsoft/MMdnn." https://github.com/microsoft/MMdnn/issues/297. Accessed 13 Dec. 2023.

[274] "How to modify the convolution property to same. - issue #135 - gmalivenko/onnx2keras." https://github.com/gmalivenko/onnx2keras/issues/135. Accessed 13 Dec. 2023.

[275] "Hyperparameter values forcefully converted to strings, thus unable to pass a list - issue #613 - aws/sagemaker-python-sdk." https://github.com/aws/sagemaker-python-sdk/issues/613. Accessed 13 Dec. 2023.

[276] ACervantes, "Strides problem on the nvconverter," 2021. Accessed 13 Dec. 2023.

[277] "Batch Normalization Layers Not Present in the Converted Keras Model - Issue #135 - Gmalivenko/pytorch2keras." https://github.com/gmalivenko/pytorch2keras/issues/135. Accessed 13 Dec. 2023.

[278] "ValueError: Graph has cycles - issue #2246 - onnx/tensorflow-onnx." https://github.com/onnx/tensorflow-onnx/issues/2246. Accessed 13 Dec. 2023.

[279] "Missing Shape Information for 'NonZero' Node Derived from 'Where' Node - Issue #1203 - Onnx/tensorflow-onnx." https://github.com/onnx/tensorflow-onnx/issues/1203. Accessed 13 Dec. 2023.

[280] "Poor TensorFlow Lite accuracy in Android application." https://stackoverflow.com/questions/69352192/poor-tensorflow-lite-accuracy-in-android-application. Accessed 13 Dec. 2023.

[281] "TVM Debugger." https://tvm.apache.org/docs/arch/debugger.html, 2023. [Accessed 13-Dec-2023].

[282] E. O. et al., "Methodology and Application of the Kruskal-Wallis Test," Applied Mechanics and Materials, vol. 611, pp. 115 – 120, 2014.

[283] B. Recht, R. Roelofs, L. Schmidt, and V. Shankar, "Do ImageNet Classifiers Generalize to ImageNet?," 2019.

[284] "ONNXMLTools - GitHub." https://github.com/onnx/onnxmltools, 2023. [Accessed 8-Dec-2023].

[285] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, ACL '02, (USA), p. 311–318, Association for Computational Linguistics, 2002.

[286] P. Christen, D. J. Hand, and N. Kirielle, "A Review of the F-Measure: Its History, Properties, Criticism, and Alternatives," ACM Comput. Surv., vol. 56, Oct. 2023.

[287] "Joblib." https://joblib.readthedocs.io/en/latest/parallel.html, 2024. [Accessed 9-Jul-2024].

[288] N. Louloudakis and A. Rajan, "OODTE: A Differential Testing Engine for the ONNX Optimizer," 2025.

[289] M. G. Kendall, "A New Measure of Rank Correlation," Biometrika, vol. 30, no. 1/2, pp. 81–93, 1938.

[290] X. Zhang et al., "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in 2018 IEEE/CVF CVPR, pp. 6848–6856, 06 2018.

[291] M. Tan et al., "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," 2020.

[292] G. Huang et al., "Densely Connected Convolutional Networks," CoRR, vol. abs/1608.06993, 2016.

[293] V. Pallipuram et al., "A comparative study of GPU programming models and architectures using neural networks," TJS, vol. 61, 09 2011.

[294] L. Vespa, "Unraveling the Divergence of GPU Threads," in CSCI '18, pp. 1398–1403, 2018.

[295] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2704–2713, 2018.

[296] Z. Liu et al., "Post-Training Quantization for Vision Transformer," 2021.

[297] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," 2021.

[298] E. H. Bersoff, V. D. Henderson, and S. G. Siegel, "Software configuration management," SIGMETRICS Perform. Eval. Rev., vol. 7, p. 9–17, Jan. 1978.

[299] C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-Based Generation of Software Product Line Test Configurations," in Search-Based Software Engineering (C. Le Goues and S. Yoo, eds.), (Cham), pp. 92–106, Springer International Publishing, 2014.

[300] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl, "Variability Issues in Software Product Lines," in Revised Papers from the 4th International Workshop on Software Product-Family Engineering, PFE '01, (Berlin, Heidelberg), p. 13–21, Springer-Verlag, 2001.

[301] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Comput., vol. 9, p. 1735–1780, Nov. 1997.