Karlsruhe Institute of Technology

Functional Testing of Deep Neural Networks

Zur Erlangung des akademischen Grades einer

DOKTORIN DER INGENIEURWISSENSCHAFTEN

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation

von
Dina Moussa
aus Cairo, Egypt

Tag der mündlichen Prüfung: 17. June 2026

1. Referent: Prof. Dr. Mehdi B. Tahoori
Karlsruhe Institute für Technology (KIT)

2. Referent: Prof. Dr. Rolf Drechsler
Universität Bremen

Acknowledgement

As I reflect on this PhD journey, I am filled with gratitude for the many people who supported me, believed in me, and stood by me throughout this chapter of my life.

First, I would like to express my sincere gratitude to my supervisor, Professor Mehdi Tahoori, for his trust, guidance, and for giving me the opportunity to be his PhD student. I have learned a lot during this journey and am still learning thanks to his support, encouragement, and confidence in me.

My deepest thanks go to my family: my father, Professor Abdelfattah Moussa, my mother, Dr. Faiza Selim, and my brother, Eng. Mohammed Moussa, for their constant love, care and unwavering support. Your belief in me, your kindness, and all the beautiful things you have given me have truly strengthened me and made my heart very happy.

A very special and heartfelt thank you goes to my husband Eng. Mohammed Bakr for his endless love, patience, and understanding throughout this demanding journey. He stood by me through the hardest moments, believed in me even when I doubted myself, and made countless sacrifices so that I could continue moving forward.

I am also endlessly grateful to my little boy, Adam, who gave me motivation every day to continue. Your smile and presence have been my greatest source of energy, and I hope you will always be proud of your mama.

I would like to extend my heartfelt appreciation to Dr. Michael Hefenbrock, my teammate and colleague, from whom I learned so much. His support made many challenges and problems easier not only in work-related matters but also through genuine friendship and guidance. He has been a great teacher and a true friend.

I also express my deep gratitude to Dr. Hassan Nassar for his continuous support and encouragement throughout my PhD journey. From the very beginning, he treated me like a sister and was always willing to help whenever I faced difficulties, both academically and personally.

I also sincerely thank Maryam Ghasemi, my colleague and dear friend, for her support, kindness, and understanding. She stood by me during some of the most difficult times in my life, always offering encouragement and friendship when I needed it the most.

My sincere thanks also go to our wonderful secretary, Iris, for her continuous support, and to our CDNC chair and colleagues for their encouragement and assistance—especially Haneen, Mahta, and Chris. I would particularly like to thank Haneen for her kindness, support, and the many moments of encouragement she shared with me, which made the journey more pleasant and enjoyable.

Finally, I would like to express my sincere appreciation to my university in Egypt, AAST (Arab Academy for Science, Technology, and Maritime Transport), for their support throughout my PhD journey.

Without all of you, this would not have been possible.

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben haben und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Karlsruhe, 15. April 2026
Dina Moussa

Abstract

Deep Neural Networks (DNNs) are increasingly deployed in safety-critical systems, where inference is performed on specialized hardware accelerators such as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), and emerging Artificial Intelligence (AI)-specific circuits. Although these platforms are typically validated through structural manufacturing tests, faults that escape production screening or arise in the field due to aging, voltage noise, radiation-induced bit flips, or device-level degradation can silently alter the functional behavior of deployed models.

Because the functionality of DNNs emerges from distributed, nonlinear computations rather than explicit Boolean specifications, conventional logic-oriented testing is insufficient to guarantee correct inference. This thesis addresses this gap by developing a functional testing methodology, where faults are detected through observable input–output behavior.

The proposed approach is based on generating functional test patterns—inputs that amplify behavioral discrepancies between a fault-free model and its faulty realization. The methodology is first developed through optimization-based techniques to generate effective test patterns and reduce test set size through compaction. It is then extended to more realistic fault conditions by targeting multiple faults simultaneously and enabling generalization to previously unseen fault instances.

To address non-differentiable objectives, such as minimizing test set size, evolutionary optimization is incorporated, enabling the handling of discrete and non-differentiable optimization problems beyond the scope of gradient-based methods. To further support deployment in resource-constrained environments, storage-efficient techniques are introduced, including test pattern compression using basis vector representations and a Neural Built-In Self-Test (NBIST) architecture that enables in-field test generation with significantly reduced memory overhead.

A unified probabilistic framework is then developed in which faults are modeled as samples drawn from a probability distribution and test patterns are generated through joint optimization to maximize expected fault coverage. This formulation eliminates the need for explicit fault enumeration and provides a principled and scalable foundation for functional testing under realistic conditions.

Finally, the methodology is extended beyond fault detection to functional equivalence checking, where in-distribution counterexamples are generated to identify behavioral discrepancies between original and transformed models, such as those obtained through quantization or pruning.

Together, these contributions establish a comprehensive and scalable approach to functional validation, bridging classical Automatic Test Pattern Generation (ATPG) principles with the reliability requirements of modern DNN accelerators in both manufacturing and in-field operation.

Zusammenfassung

Tiefe neuronale Netze (Deep Neural Networks, DNNs) werden zunehmend in sicherheitskritischen Systemen eingesetzt, in denen die Inferenz auf spezialisierten Hardware-Beschleunigern wie Graphics Processing Units (GPUs), Tensor Processing Units (TPUs) und auf neuen Schaltungen speziell für Künstliche Intelligenzen (KI) erfolgt. Obwohl diese Plattformen in der Regel durch strukturelle Fertigungstests validiert werden, können Fehler, die der Produktionsprüfung entgehen oder im Feld aufgrund von Alterung, Spannungsschwankungen, strahlungsinduzierten Bitfehlern oder gerätebedingter Degradation entstehen, das funktionale Verhalten der eingesetzten Modelle unbemerkt verändern.

Da die Funktionalität von DNNs aus verteilten, nichtlinearen Berechnungen hervorgeht und nicht auf expliziten booleschen Spezifikationen basiert, sind klassische, logikorientierte Testverfahren nicht ausreichend, um eine korrekte Inferenz zu gewährleisten. Diese Arbeit adressiert diese Lücke durch die Entwicklung einer Methodik zum funktionalen Testen, bei der Fehler anhand des beobachtbaren Ein- und Ausgabeverhaltens erkannt werden.

Der vorgeschlagene Ansatz basiert auf der Generierung funktionaler Testmuster, d. h. Eingaben, die Verhaltensunterschiede zwischen einem fehlerfreien Modell und seiner fehlerbehafteten Realisierung verstärken. Die Methodik wird zunächst mithilfe optimierungsbasierter Verfahren entwickelt, um effektive Testmuster zu erzeugen und die Größe des Testsets durch Kompaktierung zu reduzieren. Anschließend wird sie auf realistischere Fehlerszenarien erweitert, indem mehrere Fehler gleichzeitig berücksichtigt und eine Generalisierung auf zuvor nicht beobachtete Fehlerinstanzen ermöglicht wird.

Um nicht differenzierbare Zielgrößen, wie zum Beispiel die minimierung von Testsetgrößen zu behandeln, werden evolutionäre Optimierungsverfahren eingesetzt, die die Behandlung diskreter und nicht differenzierbarer Optimierungsprobleme ermöglichen, die mit gradientenbasierten Methoden nicht erkennbar sind. Zur Unterstützung des Einsatzes in ressourcenbeschränkten Umgebungen werden zudem speichereffiziente Techniken eingeführt, darunter die Kompression von Testmustern mittels Basisvektorenrepräsentationen sowie eine Neural Built-In Self-Test (NBIST)-Architektur, die eine Testgenerierung im Feld bei deutlich reduziertem Speicherbedarf ermöglicht.

Ein einheitliches probabilistisches Framework wurde entwickelt, in welchem Fehler als eine Stichprobe aus einer Wahrscheinlichkeitsverteilung modelliert werden und Testmuster durch gemeinsame Optimierung generiert werden, um die erwartbare Fehlerabdeckung zu maximieren. Diese Formulierung macht eine explizite Enumeration aller Fehler überflüssig und bietet eine fundierte sowie skalierbare Grundlage für funktionales Testen unter realistischen Bedingungen.

Zum Schluss wird die Methodik über die reine Fehlererkennung hinaus auf funktionale Äquivalenzprüfung erweitert, bei der In-Distribution-Gegenbeispiele generiert werden, um Verhaltensunterschiede zwischen ursprünglichen und transformierten Modellen, etwa durch Quantisierung oder Pruning, zu identifizieren.

Insgesamt etablieren diese Beiträge einen umfassenden und skalierbaren Ansatz zur funktionalen Validierung, der klassische Prinzipien der Automatic Test Pattern Generation (ATPG) mit den Zuverlässigkeitsanforderungen moderner DNN-Beschleuniger sowohl in der Fertigung als auch im Feldeinsatz verbindet.

Contents

Part I.

Preliminaries

1. Introduction

1.1. Motivation: Why Functional Testing for Deep Neural Networks?

Deep Neural Networks (DNNs) have become a foundational technology in modern computing, enabling high-accuracy perception, prediction, and decision-making [69]. Their impact is particularly evident in domains such as autonomous driving, healthcare, robotics, industrial inspection, and intelligent edge devices, where DNNs are increasingly involved in safety-relevant decisions [70]. In many of these deployments, inference must satisfy strict latency and energy constraints, accelerating the shift toward specialized AI Hardware Accelerators (AI-HAs). This trend has driven research into architectures such as memristive crossbars [40] and large-scale commercial systolic arrays [2] (e.g., Google Tensor Processing Unit (TPU) [33]). This specialization is now widespread, with dedicated accelerators deployed in cloud data centers (e.g., NVIDIA tensor-core Graphics Processing Units (GPUs) [42], AWS Trainium [89]) and at the edge (e.g., Google Edge TPU [66], mobile TPU [67] in modern system on a chips (SoCs)).

Despite these advances, reliability assurance for DNN inference remains a major challenge [44]. Neural inference hardware is subject to a range of failure mechanisms, including manufacturing defects that escape structural tests, permanent faults caused by wear-out and aging (e.g., bias temperature instability, hot-carrier injection), parametric variations due to voltage and temperature fluctuations, and transient soft errors (e.g., radiation-induced bit flips). These faults often do not produce catastrophic failures, such as system crashes [23], [60]. Instead, they manifest themselves as subtle numerical perturbations inside the computation pipeline, which can lead to silent accuracy degradation, output misclassifications, or safety-critical decision errors [40], [58], [61].

A key difficulty is that conventional hardware testing was designed for Boolean logic circuits and explicitly specified functions [14]. Structural test approaches, such as stuck-at and transition fault testing, aim to activate a fault at an internal site and propagate it to an observable output. However, for DNN inference, functionality emerges from a sequence of nonlinear transformations distributed across layers, weights, and activations, and the accelerator often exposes only high-level outputs [11], [40]. Consequently, structural methods alone do not provide sufficient confidence that the deployed network preserves its intended learned behavior. This mismatch motivates functional testing, i.e., validating system correctness based on input-output behavior rather than internal circuit states [46].

1.2. What Does Functionality Mean for Deep Neural Networks?

For classical digital hardware, a functional specification is typically a truth table or an algorithm that defines the correct behavior for all inputs [14]. In contrast, a DNN defines a learned mapping from a high-dimensional input space (e.g., images, signals, or multimodal sensor inputs) to outputs (e.g., class labels, continuous values, or control actions) [37]. For a classifier, the deployed functionality can be expressed as

$$f(x) = \arg\max \mathrm{Net}(x),$$

where $\mathrm{Net}(x)$ denotes the output score (logit or probability), and arg max returns the index of the class with the largest score. Under faults, the accelerator effectively implements a perturbed function $f_f(x)$ induced by a faulty realization $\mathrm{Net}_f(x)$. The central question becomes: does the faulty implementation preserve the intended mapping sufficiently to be considered correct?

From a reliability perspective, the most critical events are those where faults induce observable behavioral changes, such as decision flips. Unlike small numerical differences in internal tensors (which may be unobservable in practice), a label change is directly meaningful to the system and can be checked even when only a black-box interface is available. This thesis therefore focuses on functional testing criteria that are both practically observable and operationally relevant [50].

1.3. Functional Test Patterns: Concept and Role

Functional testing aims to detect faults by applying carefully selected inputs and monitoring the resulting outputs [55]. A functional test pattern is an input designed to amplify the effect of faults so that deviations become observable at the output. In the DNN setting, let $\mathrm{Net}(x)$ be the fault-free model and $\mathrm{Net}_f(x)$ its faulty realization. Under a realistic black-box assumption, a test pattern detects a fault if it changes the predicted label:

$$\arg\max \mathrm{Net}(x) \;\neq\; \arg\max \mathrm{Net}_f(x).$$

This criterion captures a functional mismatch at the decision level and avoids relying on internal signals or intermediate activations.

The need for targeted pattern generation arises because the faults in DNNs are frequently masked [47]. Redundancy in weights and channels, nonlinearities such as Rectified Linear Units (ReLUs), normalization layers, and the statistical robustness of learned representations can suppress fault effects [46]. As a result, many random inputs will not expose a given fault, especially if the fault becomes visible only in rarely activated regions of the input space. Effective functional testing therefore requires systematic generation of test patterns that effectively explore sensitive regions near the model's decision boundaries, where small perturbations are more likely to induce observable output changes.

1.4. Desired Properties of Functional Test Sets

A test set is not useful merely because it detects faults. In practice, testing is constrained by time, storage, and observability. This motivates several key properties:

- **Fault coverage:** the fraction of modeled faults that are detected by the test set.
- **Compactness:** a small number of test patterns to reduce test application time and cost.
- **Storage efficiency:** minimal memory usage, particularly important for in-field or on-device testing.
- **Scalability:** effectiveness in modern networks, including deep Convolutional Neural Networks (CNNs) and large models, not limited to small-scale architectures.
- **Accessibility (black-box operation):** reliance only on input-output behavior to support sealed accelerators and Intellectual Property (IP) constraints.

These properties are inherently interdependent. For example, maximizing coverage can lead to a large number of test patterns unless redundancy is controlled. Similarly, reducing storage may require compressed representations or on-demand generation, which in turn affects how test patterns are generated and applied. A central theme of this thesis is that functional testing methods must be designed with these constraints explicitly in mind from the outset.

1.5. Challenges Unique to Deep Neural Networks Functional Testing

Functional testing for DNN accelerators differs fundamentally from classical testing in several ways.

1.5.1. High-dimensional input spaces and complex decision boundaries

DNN inputs live in high-dimensional spaces, for example, a 224×224 RGB image contains more than 150,000 input values, meaning that even small perturbations can move an input in many directions. Moreover, the classifier decision boundary is highly nonlinear and strongly shaped by the training data, resulting in regions where the predicted label can change that are irregular and difficult to characterize analytically. As a result, finding fault-exposing patterns is not a simple search problem. It becomes a challenging optimization task that requires navigating a vast input space to locate test patterns.

1.5.2. Fault modeling

Although physical faults originate at the circuit level (e.g., memory-cell defects, transient storage malfunctions, or variations in arithmetic), the generation of functional tests requires manageable abstractions that describe how such effects appear at the level of DNN computation. In this thesis, we focus on fault models that directly capture the dominant ways inference can be corrupted through network parameters: bit flips, stuck-at faults, weight perturbations, and composite combinations of these effects.

A weight bit flip models an error in the binary representation of a stored weight, as can occur due to transient soft errors or unreliable memory reads. In quantized networks, flipping even a single bit can cause a noticeable change in the represented value, potentially amplifying into a decision error. A stuck-at fault models a weight element (or a bit within its representation) that is permanently forced to a fixed value (0 or 1), reflecting manufacturing defects or wear-out mechanisms; unlike bit flips, this corruption persists across all inferences. A weight perturbation models deviations where weights drift from their nominal values by an additive or multiplicative distortion, capturing effects such as parametric variation, retention loss, or analog noise; this model is useful when errors are not well described as discrete bit-level events but rather as continuous deviations.

In practice, faults may not occur in isolation. Therefore, we also consider composite faults where multiple corruptions coexist, for example, a stuck-at defect in one region of memory combined with occasional transient bit flips elsewhere, or permanent stuck-at behavior together with small perturbations caused by operating-condition changes. Such composite modeling is important because the observable behavior of the accelerator is determined by the combined effect of all active error sources.

A key challenge is that even these seemingly simple models can produce a very large fault space (e.g., many weights, many bit positions, many possible fault locations). Exhaustive enumeration is rarely feasible. Functional testing therefore relies on tractable fault sets, typically obtained through structured selection (e.g., targeting representative layers or regions) or statistical sampling, while ensuring that the modeled faults remain realistic and relevant to deployment.

1.5.3. Masking and non-uniform sensitivity

The effects of the fault in DNNs are often masked, which means that internal corruption does not necessarily change the final decision. This happens because DNNs often contains redundancy and distributed representations: multiple weights and channels can contribute similar evidence, so an error in one component may be compensated for by others. In addition, nonlinearities and normalization can suppress or reshape errors; for example, if a corrupted computation lies in a pathway that is weakly activated for a given input, the resulting deviation may be too small to affect the output label.

Sensitivity is also non-uniform across the network. Corrupting different weights can have very different impacts: some weights lie in critical pathways that strongly influence class evidence, while others are effectively redundant or rarely influential. This non-uniformity appears across layers and channels as well: certain layers or feature channels may be much more important for the final decision, and faults there are more likely to cause observable misbehavior.

Crucially, sensitivity is input-dependent. The same faulty weight can be invisible for many inputs but highly detectable for specific inputs that activate the affected pathway or operate close to a decision boundary. This dependence means that random inputs often fail to expose faults because they may not trigger vulnerable computations or may produce large classification margins that tolerate moderate internal perturbations.

These observations motivate coverage-oriented and diversity-driven functional tests. To achieve high fault coverage, test patterns should be generated to activate different internal pathways across the model and probe inputs where decisions are more sensitive to perturbations. Consequently, an effective test set is not just a collection of arbitrary samples but a deliberately constructed pattern whose diversity increases the likelihood that at least one pattern exposes each modeled fault through an observable decision change.

1.5.4. Deployment constraints: black-box interfaces and in-field testing

In manufacturing, testing can exploit Design-For-Tests (DFTs) infrastructure such as scan chains and internal debug visibility, allowing engineers to directly control and observe internal states and apply classical structural test techniques. After deployment, DNN accelerators are often delivered as sealed IP blocks or secure modules, so the tester typically has access only to the external input-output interface: intermediate activations, feature maps, and partial sums are hidden, and even confidence information such as logits may be unavailable, leaving only the final predicted label as an observable signal.

At the same time, in-field testing must run under tight operational budgets: it must be fast enough to fit into short idle or maintenance windows, lightweight enough to avoid extra instrumentation or heavy computation beyond normal inference, memory-efficient enough to avoid storing large test libraries on resource-constrained devices, and non-disruptive so that it does not interfere with normal system operation. These constraints strongly favor functional methodologies that can detect faults robustly using only black-box input-output behavior, with compact test data and simple decision-level pass/fail criteria.

1.6. Thesis Vision and Approach

This thesis develops a unified approach to functional testing that adapts ideas from the classical Automatic Test Pattern Generation (ATPG) domain to the DNN domain while respecting the specific constraints of DNNs. The core philosophy is the following.

> *Generate compact functional test patterns that expose modeled faults through observable input-output behavior and make testing deployable by explicitly addressing scalability and storage.*

To achieve this, the thesis builds progressively across multiple dimensions.

1. **More realistic testing:** generate test patterns that remain effective under practical hardware fault scenarios, not limited to single-fault scenarios.
2. **Smaller test sets:** generate compact test patterns from the beginning, avoiding the need to first create a large test set and then heavily compact them.
3. **Lower memory cost:** minimize the storage required for testing and support generating or regenerating patterns on demand instead of storing all patterns explicitly.
4. **Beyond hardware faults:** apply functional testing to verify that model transformations (e.g., pruning or quantization) preserve the intended input-output behavior.

1.7. Contributions and Research Directions

This thesis presents a comprehensive framework for functional test pattern generation for DNN accelerators, systematically adapting principles from classical ATPG to modern AI hardware. The contributions are summarized below.

1.7.1. Functional Automatic Test Pattern Generation for Deep Neural Networks

A functional ATPG methodology is developed in which fault models are defined at the level of neurons and weights. Test generation is formulated as an optimization problem that seeks inputs that maximize behavioral divergence between fault-free and faulty models, with a focus on decision-level observability. To reduce redundancy and cost, compaction strategies are introduced, including a heuristic algorithm that eliminates overlapping patterns and cluster-based representative selection that preserves fault diversity while reducing test set size. Through these contributions, functional ATPG is established as a practical and principled baseline for DNNs reliability analysis [80].

1.7.2. Compact Functional Testing Under Multiple Faults

To reflect realistic fault conditions, test generation is performed with respect to a fault list instead of optimizing patterns for a single fault at a time. Each generated pattern is therefore encouraged to detect as many faults as possible from the target list, which naturally leads to compact test patterns from the start instead of relying on heavy post-generation compaction. This fault-list-driven formulation also scales better to large fault spaces, since evaluation and optimization are performed at the level of fault coverage over the selected list. A key benefit is generalization: patterns optimized for representative sampled faults often remain effective for additional, unseen faults because they concentrate on sensitive behaviors and decision-boundary regions where many different faults tend to become observable [81], [90].

1.7.3. Evolutionary Optimization for Non-differentiable Objectives

While gradient-based approaches are well suited for differentiable objectives (e.g., maximizing output deviation), minimizing test pattern size is inherently non-differentiable. Therefore, evolutionary optimization methods (e.g., Covariance Matrix Adaptation Evolution Strategys (CMA-ESs)) are employed to directly search for compact sets of test patterns that maximize fault coverage. Through this approach, a population of candidate test sets is iteratively evolved, naturally balancing coverage and compactness without requiring differentiable relaxations [96].

1.7.4. Storage-efficient Test Generation and In-field Self-test

Beyond the size of the test pattern, storage constraints are a dominant bottleneck for in-field testing. Compressed test patterns are generated where the patterns are represented as linear combinations of basis vectors, dramatically reducing memory requirements while maintaining high coverage [93]. In addition, an in-field testing architecture is proposed that generates tests on demand under strict storage requirements. In particular, Neural Built-In Self-Tests (NBISTs) use a compact generator network that maps pseudo-random seeds to structured high-coverage functional patterns, enabling self-test during deployment with significantly reducing memory requirements [98].

1.7.5. Unified Framework for In-field Testing

The thesis introduces a unified approach that can combine different fault models within a single methodology. Using Monte Carlo sampling and probabilistic mixtures of permanent and transient errors,

test generation maximizes expected coverage under realistic fault distributions. Importantly, all patterns are generated at once under explicit storage constraints, making memory usage predictable and allowing systematic trade-off analysis between coverage, test pattern size, and storage [95].

1.7.6. Equivalence Testing of Model Transformations

Finally, functional validation extends beyond faults to model transformations, such as pruning and quantization of models. Since accuracy alone may fail to capture rare but important behavioral changes, equivalence checking based on generating in-distribution counterexamples is proposed to reveal non-equivalence between original and transformed models [94].

1.7.7. Discussion and Insights

Across all contributions, four main insights emerge. First, when test generation is guided by a decision-level objective (i.e., focusing on whether the predicted label changes), functional ATPG for DNNs is consistently more effective than simple random testing and standard adversarial baselines. Second, no single technique is optimal under all constraints: gradient-based optimization is well suited to quickly produce fault-exposing patterns, while evolutionary search is particularly useful when the goal is to obtain compact test patterns directly during generation; additionally, compression and NBIST are essential when storage and field memory budgets become the limiting factor. Third, the overall approach demonstrates strong scalability in practice, extending from small multilayer perceptrons (MLPs) to deep CNNs without changing the core principles. Finally, because the methods rely only on observable input-output behavior, they remain applicable even when accelerators provide only black-box access in real deployments.

1.8. Dissertation Outline

This dissertation is structured as follows.

- **Chapter 2** introduces background on DNNs, their architectures, fault models, and core testing concepts such as ATPG, fault coverage, compaction, and Built-In Self-Test (BIST).
- **Chapter 3** reviews related work on DNNs reliability and functional testing.
- **Chapter 4** presents the functional ATPG framework, including optimization, fault modeling, and compaction.
- **Chapter 5** extends the framework to fault-list-driven testing and compact pattern generation.
- **Chapter 6** develops evolutionary methods for compact test generation under non-differentiable objectives.
- **Chapter 7** proposes storage-efficient testing, including compressed representations and NBIST for in-field test generation.
- **Chapter 8** presents a unified in-field testing methodology under storage constraints.
- **Chapter 9** introduces equivalence checking and counterexample generation for transformed models.
- **Chapter 10** concludes the dissertation and outlines future directions.

2. Background

2.1. Deep Neural Networks (DNNs)

2.1.1. Structure and Forward Propagation

DNNs are parametric models that learn mappings from an input space $\mathcal{X}$ to an output space. In this thesis, the focus is on supervised learning settings where the network produces an output vector in $\mathbb{R}^C$, with C denoting the number of classes.

$$\mathrm{Net}(\mathbf{x};\theta):\ \mathcal{X}\to\mathbb{R}^C.$$

The parameters θ collect all trainable weights and biases across the network

$$\theta=\{\mathbf{W}^l,\mathbf{b}^l\mid l=1,\dots,L\}.$$

A network is composed of L layers. Each layer combines the input features through an affine transformation and then applies a nonlinear activation function. For each layer l, the computation is as follows

$$\mathrm{Layer}^l(\mathbf{x}^{l-1})=\phi\left(\mathbf{x}^{l-1}\mathbf{W}^l+\mathbf{b}^l\right), \tag{2.1}$$

where $\mathbf{W}^l$ is the weight matrix, $\mathbf{b}^l$ is the bias vector, and $\phi(\cdot)$ is an elementwise activation function. The role of $\phi(\cdot)$ is essential: without nonlinear activations, the composition of affine layers would collapse to a single affine transformation, preventing the model from representing complex decision boundaries. Common activation functions include sigmoidal functions such as sigmoid and tanh, which saturate for large positive or negative inputs, and piecewise/continuous linear functions such as ReLUs and Sigmoid Linear Units (SiLUs), which are widely used in modern deep networks due to their favorable optimization behavior.

The forward propagation can also be written recursively. The input is denoted as $\mathbf{x}^0=\mathbf{x}$. For each layer l, the computation is given by

$$\mathbf{x}^l=\mathrm{Layer}^l(\mathbf{x}^{l-1}),\quad l=1,\dots,L. \tag{2.2}$$

The network output is given by

$$\mathrm{Net}(\mathbf{x})=\mathbf{x}^L. \tag{2.3}$$

This layered structure enables the network to learn hierarchical representations: early layers typically extract low-level features, while deeper layers combine them into more abstract and task-specific concepts. The final layer is often referred to as the output layer. Depending on the task, its activation may be omitted so that the network outputs logits, i.e., unnormalized class scores.

For classification, the logits are commonly converted into probabilities with the Softmax function. For a vector $\mathbf{z}\in\mathbb{R}^C$ (interpreted here as logits), the c-th entry is

$$\mathrm{Softmax}(\mathbf{z})_c:=\frac{\exp(z_c)}{\sum_{c'}\exp(z_{c'})}. \tag{2.4}$$

Softmax produces a normalized vector over classes, which is useful for interpreting outputs and is the standard interface for cross-entropy training. In segmentation tasks, the same idea is applied independently at each pixel: Softmax is computed independently at each spatial location so that each pixel is assigned a

valid class-probability vector. This allows the network to produce structured predictions (e.g., semantic maps) while still using the same probabilistic interpretation of outputs.

2.1.2. Neural Network Tasks

DNNs can be applied to different learning tasks depending on the structure of the desired output. In this thesis, the focus is on tasks where the network produces predictions for classification or structured outputs over spatial domains.

Classification. In classification, the goal is to assign an input sample $\mathbf{x}$ to one of C discrete classes. The network outputs a vector in $\mathbb{R}^C$ containing class scores (logits), which are typically converted into probabilities using the Softmax function (Eq. 2.4). The predicted class corresponds to the highest probability. Image classification datasets such as CIFAR-10 or ImageNet are common examples.

Segmentation. In segmentation tasks, a class label is assigned to each spatial element of the input, such as each pixel in an image. The network therefore produces a spatial map of class scores, and Softmax is applied independently at each location to obtain per-pixel class probabilities, forming a semantic segmentation map.

Although these tasks differ in output structure, both rely on the same forward inference pipeline. Consequently, the reliability analysis and test generation techniques developed in this thesis apply to both settings.

2.1.3. Training Objective

Training learns parameters θ from a labeled dataset

$$\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N},$$

where $\mathbf{x}_n$ denotes an input sample and $\mathbf{y}_n$ its corresponding ground-truth label. The form of $\mathbf{y}_n$ depends on the task: for classification, it typically corresponds to a class index in $\{1, \ldots, C\}$, whereas for segmentation, it represents a spatial label map. Learning is formulated as empirical risk minimization

$$\theta^\star = \arg\min_\theta \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\mathrm{Net}(\mathbf{x}_n; \theta), \mathbf{y}_n) \,. \tag{2.5}$$

where $\mathcal{L}(\cdot, \cdot)$ is a task-dependent loss function. Cross-entropy is commonly used for classification and pixel-wise cross-entropy for segmentation because these losses naturally connect to probabilistic outputs produced by Softmax. Optimization is performed by computing gradients of the loss with respect to θ via backpropagation, followed by parameter updates using algorithms such as Stochastic Gradient Descents (SGDs) [1] or Adam [27]. Once training is complete, inference evaluates $\mathrm{Net}(\mathbf{x}; \theta^\star)$ on previously unseen inputs. Since deployed systems execute inference repeatedly and often under strict energy/latency constraints, the reliability of inference under realistic hardware conditions is a central concern of this thesis.

2.1.4. Gradient-Based Optimization

The optimization problem introduced in Eq. (2.5) is typically solved using gradient-based optimization methods. These methods iteratively update the parameters θ in the direction of decreasing empirical risk.

Let the empirical loss be denoted as

$$J(\theta) := \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\mathrm{Net}(\mathbf{x}_n; \theta), \mathbf{y}_n) \,.$$

Gradient-based optimization updates the parameters according to

$$\theta^{(t+1)} = \theta^{(t)} - \eta^{(t)} \nabla_\theta J(\theta^{(t)}), \tag{2.6}$$

where $\eta^{(t)} > 0$ is the learning rate at iteration t, and $\nabla_\theta J(\theta)$ denotes the gradient of the loss with respect to all trainable parameters.

Computing the full gradient over the entire dataset can be computationally expensive for large-scale problems. Therefore, in practice, SGDs is used. Instead of evaluating the gradient over all N samples, it is approximated using a mini-batch $\mathcal{B}_t \subset \mathcal{D}$:

$$\nabla_\theta J(\theta) \approx \frac{1}{|\mathcal{B}_t|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{B}_t} \nabla_\theta \mathcal{L}(\mathrm{Net}(\mathbf{x};\theta), \mathbf{y}) \,. \tag{2.7}$$

The parameters are then updated using this stochastic estimate of the gradient. Although each update is noisy, repeated iterations converge to a local minimum under suitable conditions.

The efficient computation of $\nabla_\theta J(\theta)$ is based on backpropagation, which applies the chain rule of calculus to the layered structure of the network in Eq. (2.2).

Let the intermediate pre-activation at layer l be

$$\mathbf{z}^l = \mathbf{x}^{l-1}\mathbf{W}^l + \mathbf{b}^l,$$

and the post-activation be

$$\mathbf{x}^l = \phi(\mathbf{z}^l).$$

Given a loss $\mathcal{L}$ evaluated in the output layer, the gradients are propagated backward from layer L to layer 1. Defining the layer-wise error signal

$$\boldsymbol{\delta}^l := \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l},$$

the chain rule yields

$$\boldsymbol{\delta}^l = \left(\boldsymbol{\delta}^{l+1}(\mathbf{W}^{l+1})^\top\right) \odot \phi'(\mathbf{z}^l), \tag{2.8}$$

where $\odot$ denotes elementwise multiplication and $\phi'(\cdot)$ is the derivative of the activation function.

The gradients with respect to the parameters are given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} = (\mathbf{x}^{l-1})^\top \boldsymbol{\delta}^l, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l. \tag{2.9}$$

This recursive structure allows gradients to be computed with computational cost proportional to a single forward pass, making deep network training feasible.

In addition to classical SGDs, adaptive methods such as Adam [27] are widely used. Adam maintains exponential moving averages of first and second moments of the gradients and rescales parameter updates accordingly. Although the general update structure remains the same,

$$\theta^{(t+1)} = \theta^{(t)} - \eta^{(t)} \widehat{\nabla}_\theta J(\theta^{(t)}),$$

the effective learning rate becomes parameter-dependent. Such adaptive schemes often improve convergence speed and stability, especially in high-dimensional and non-convex optimization landscapes typical for deep learning.

Gradient-based optimization is not only central to training Neural Networks (NNs), but also plays a key role in several methods developed in this thesis. In later chapters, gradients are used to:

- Optimize test inputs to maximize functional deviation under faults,

- Search for counterexamples in equivalence checking,
- Refine compact test representations with differentiable objectives.

Therefore, understanding gradient computation and its limitations is fundamental for both model training and reliability-oriented optimization procedures considered throughout this work.

2.1.5. Common Architectures: MLPs and CNNs

A network composed of dense (fully connected) layers is referred to as a MLP. In an MLP, each neuron in a layer depends on all outputs of the previous layer, making it suitable for structured or low-dimensional data and for controlled experiments where model behavior is easier to analyze. CNN extend this design with convolutional layers that apply learnable filters across spatial dimensions. Convolutions introduce parameter sharing and locality, enabling efficient feature extraction for grid-like data such as images. Modern CNNs typically combine convolution operations with nonlinear activations and may include pooling and normalization layers to improve robustness, training stability, and efficiency. Because CNNs can be deep and contain many channels and feature maps, fault effects can depend strongly on which features are activated by a specific input, making them representative targets for functional reliability evaluation.

2.2. Testing Fundamentals

Testing ensures that hardware systems operate correctly in the presence of manufacturing defects, operational faults, or aging effects. Classical testing methodologies were developed for digital logic circuits with explicitly defined functionality, where faults are activated and propagated to observable outputs using carefully designed test patterns.

Although these principles remain applicable, testing DNN accelerators introduces additional challenges. NNs implement learned input-output mappings rather than fixed Boolean specifications, and accelerators often expose only high-level outputs. Therefore, testing must rely primarily on observable input-output behavior. The concepts introduced in this section provide the foundation for the functional testing methodologies developed in this thesis.

2.2.1. Automatic Test Pattern Generation

ATPG is a classical testing methodology used to automatically generate input patterns that detect modeled faults in a circuit. The objective of ATPG is to activate a fault at an internal location and propagate its effect to an observable output so that the presence of the fault can be detected.

In traditional digital circuits, ATPG algorithms exploit the structure of the logic network to systematically construct such patterns. In the context of DNNs, however, internal signals are often not accessible, particularly when accelerators are deployed as sealed IP blocks. Therefore, this thesis adapts the principles of ATPG to a functional setting, where patterns are generated by reasoning about the behavioral difference between fault-free and faulty network executions.

2.2.2. Fault Coverage

Fault coverage measures the effectiveness of a test set in detecting modeled faults. It is defined as the ratio between the number of detected faults and the total number of considered faults. Let $\mathcal{F}$ denote the set of modeled faults and $\mathcal{F}_{\text{det}} \subseteq \mathcal{F}$ the subset of faults detected by the applied test set. The fault coverage is defined as

$$\text{Fault Coverage} = \frac{|\mathcal{F}_{\text{det}}|}{|\mathcal{F}|}.$$

High fault coverage indicates that the test set is effective in exposing the modeled faults. In functional testing of DNNs, a fault is considered detected if at least one test pattern produces an observable behavioral deviation between the fault-free and faulty models, typically in the form of a decision flip in the predicted label.

2.2.3. Test Compaction

While achieving high fault coverage is important, practical testing must also consider test application time and storage overhead. Large test sets increase the time required to execute tests and may exceed the available storage capacity, particularly in in-field testing scenarios.

Test compaction techniques aim to reduce the number of test patterns while maintaining high fault coverage. Two main strategies are commonly employed. Static compaction removes redundant patterns after test generation by identifying patterns whose detected faults are already covered by other patterns. Dynamic compaction incorporates coverage considerations directly during test generation, encouraging each pattern to detect multiple faults simultaneously.

In the context of DNN functional testing, compaction is particularly important because the input space is high-dimensional and naive test generation can produce large numbers of patterns. Many techniques proposed in this thesis therefore integrate compaction objectives directly into the generation process.

2.2.4. Built-In Self-Test

BIST refers to testing techniques where a hardware system includes on-chip mechanisms that allow it to test itself without relying on external testing equipment. A typical BIST architecture consists of a test pattern generator, the circuit under test, and a response analyzer that determines whether the observed outputs match expected behavior.

BIST is particularly attractive for in-field testing because it enables periodic health checks during deployment. However, implementing BIST for DNN accelerators presents challenges related to storage and computational overhead, especially when large test libraries are required.

To address these constraints, modern approaches often employ compact pattern generators or compressed test representations. The testing architectures developed later in this thesis extend this idea by generating functional test patterns on demand, enabling efficient in-field testing with minimal memory overhead.

2.3. Hardware Fault Abstractions for Neural Networks

2.3.1. Fault Transformations and Probabilistic Modeling

Although physical defects originate at the circuit level, reliability analysis and test generation require abstractions that can be applied at the model level. In this thesis, hardware-induced faults are modeled as transformations that map the fault-free model to a faulty model:

$$f : \text{Net} \;\mapsto\; \text{Net}_f.$$

This view is convenient because it directly connects hardware effects to the computation performed during inference. Instead of modeling each defect site in the underlying circuit, the transformation f

captures how the effective model used for inference differs from the intended one. Depending on the fault mechanism, the transformation may represent a deterministic modification (e.g., a permanent defect) or a stochastic modification (e.g., a transient upset).

To unify these cases, a probability distribution $p(f)$ over fault transformations is assumed. The sampling of a concrete fault instance is denoted by $f^{(n)} \sim p(f)$, and its application to the model yields $\text{Net}_{f^{(n)}}$. This probabilistic perspective allows two common evaluation modes to be considered: (i) specific faults drawn from $p(f)$ are analyzed to estimate expected reliability, and (ii) a discrete fault list $\mathcal{F}$ is defined that contains representative transformations, which can be interpreted as a discrete distribution over a finite set of fault operators.

2.3.2. Weight Perturbation, Bit Flips, Stuck-at, and Composite Faults

This thesis focuses on parameter-level fault mechanisms because weights dominate the computation and memory consumption of inference accelerators and are common points of vulnerability. The models below describe how the parameter set θ is modified under different fault mechanisms.

Weight perturbation transformations. Weight perturbation models continuous deviations in parameters that arise from hardware fluctuations, noise, or analog variation. A common abstraction is additive Gaussian noise:

$$f_{\text{perturb}}(\text{Net}) : \begin{cases} \text{Perturb network parameters } \theta \text{ by} \\ \theta \leftarrow \theta + \boldsymbol{\epsilon} \quad \text{with} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \cdot \mathbf{I}) \end{cases}$$

The parameter σ controls the severity of the fault: the larger σ corresponds to stronger deviations from the trained weights. This model is particularly useful when errors cannot be cleanly described as discrete events but rather as small, distributed changes across many parameters. In some settings, perturbations are modeled multiplicatively, e.g., $\theta \leftarrow \theta \cdot \exp(\boldsymbol{\epsilon})$, which preserves the sign of θ and can better reflect proportional drift.

Bit flip transformations. Bit flip faults model transient upsets in the binary representation of stored parameters. These faults are common abstractions for soft errors in memory or datapaths, where a bit changes value unexpectedly. The transformation is modeled as follows:

$$f_{\text{bitflip}}(\text{Net}) : \begin{cases} \text{Flip each bit in } \theta \text{ with probability } \rho \\ \theta \leftarrow \text{FlipBits}(\theta, \rho) \end{cases}$$

Here, ρ defines the bit error rate. Even a small ρ can have a noticeable effect in quantized models because each bit contributes directly to the represented value. Bit flips can be injected uniformly across parameters or restricted to specific memory regions depending on the assumed hardware scenario.

Stuck-at transformations. Stuck-at faults (SAF) represent permanent defects in which the parameters are forced to a constant value v (often 0). Such faults can arise from manufacturing defects or wear-out in memory cells. At the parameter level, a simple abstraction is

$$f_{\text{stuck@v}}(\text{Net}) : \begin{cases} \text{Fix } \rho \text{ percent of parameters to } v \\ \theta_i \leftarrow v \quad \text{if} \quad \text{Bernoulli}(\rho) = 1 \text{ (independently for each } i) \end{cases}$$

This model captures persistent corruption: once a parameter is stuck, it remains corrupted across all inferences. Choosing $v = 0$ corresponds to stuck-at-zero behavior, which effectively removes contributions of the affected weights.

Composite transformations. Real deployments may exhibit multiple fault mechanisms simultaneously, for example persistent defects combined with transient upsets or continuous variations caused by operating conditions. We model such cases through composition of fault transformations, for example,

$$f_{\text{composite}}(\text{Net}) = f_{\text{perturb}} \circ f_{\text{bitflip}}(\text{Net}).$$

Composite modeling is important because the resulting behavior is determined by the combined effect on the set of parameters θ, and interactions between mechanisms can increase the probability of functional failures compared to each mechanism in isolation.

3. Related Work

3.0.1. Structural and Functional Testing of AI Accelerators

The testing of AI-HAs can generally be divided into structural and functional approaches. Structural testing focuses on validating the underlying hardware independently of the NNs model executed on it. It is typically applied during manufacturing using well-established methods such as ATPGs, scan testing, and BISTs, targeting components including arithmetic logic units, register files, interconnects, and memory arrays. Previous work has explored these techniques for GPUs, TPUs, systolic arrays, and emerging accelerator architectures [79], [83], [84], [88]. These methods are effective at detecting fabrication defects and classical stuck-at or transition faults at the logic level.

However, structural testing alone is insufficient to guarantee the correct behavior of deployed DNNs. A DNNs accelerator implements a learned high-dimensional nonlinear function rather than a precisely specified Boolean function. Faults that escape structural tests or occur during field operation (e.g., aging-induced defects, soft errors, or parametric variations) may manifest as subtle numerical deviations that do not violate structural correctness but still alter the inference outcome. Consequently, functional testing has emerged as a complementary paradigm, focusing on validating the correctness of the accelerator by observing input–output behavior under realistic workloads [46], [77], [91].

Functional testing for DNNs aims to detect hardware-induced faults by applying carefully designed input patterns and observing the resulting input–output behavior. Unlike structural methods, which rely on the controllability and observability of internal states, functional approaches often operate under limited visibility and may assume only black-box access to the accelerator interface. This distinction fundamentally shapes the design of test generation strategies.

3.0.2. Selection-Based Functional Testing

Selection-based approaches attempt to identify fault-sensitive inputs from existing datasets. Rather than generating new patterns, these methods rank or filter available samples based on their sensitivity to injected faults.

For example, Hsieh et al. [52] used Monte Carlo simulations to evaluate the robustness of Spiking Neural Networks (SNNs) under neuron and synaptic-level faults. By comparing the $F1$ score of faulty models against a fault-free baseline, they identified performance degradation caused by connection-related faults. The complete dataset was effectively treated as a set of test patterns, and sensitivity was assessed statistically.

Similarly, Meng et al. [68] proposed a sensitivity-driven selection method in which images are ranked according to how frequently their predictions change under different fault injections. Inputs exhibiting high sensitivity are selected as test patterns. This approach achieves a low false negative rate and efficiently detects certain types of faults.

Although selection-based methods are computationally efficient and easy to deploy, they are fundamentally limited by their dependence on pre-existing datasets. The diversity and representativeness of the test patterns are bounded by the dataset distribution, restricting the ability to explore unexplored or rare regions of the input space where certain faults may become observable.

3.0.3. Mutation-Based and Adversarial Approaches

Mutation-based approaches modify existing inputs to amplify fault effects. A prominent line of work leverages adversarial example generation techniques originally developed for robustness analysis.

Goodfellow et al. [26] introduced the Fast Gradient Sign Methods (FGSMs), which perturbs an input x in the direction of the gradient of the loss function, where we follow the original notation of [26], using x for the input, θ for the parameters, and $L(\cdot, \cdot)$ for the loss:

$$\eta = \epsilon \cdot \text{sign}\left(\nabla_x L_\theta(x, y)\right) \tag{3.1}$$

where ϵ controls the perturbation magnitude. Although originally intended to produce imperceptible adversarial examples, this technique has been adopted for functional fault detection.

Wen Li et al. [45] applied adversarial image generation to detect random stuck-at faults in the bit representation of network weights. The presence of faults was determined by observing the output variation between fault-free and faulty models.

These approaches exploit the model sensitivity to input perturbations and can effectively expose certain weight-level faults. However, they typically rely on small perturbations of existing dataset samples, thereby limiting exploration of the input space. Moreover, these approaches can result in large test sets, increasing storage requirements and test application time. Because adversarial methods are designed to preserve perceptual similarity, they may constrain the diversity of generated patterns, potentially limiting fault coverage.

3.0.4. Gradient-Based Functional Test Generation

Beyond adversarial mutation, several works explicitly formulate test generation as an optimization problem targeting fault observability.

Luo et al. [46] proposed a gradient-based functional test generation approach to validate IPs of DNNs under malicious perturbations. By optimizing input perturbations, they generated patterns that amplify the difference between fault-free and faulty networks.

Qi Liu et al. [56] introduced two fault detection strategies addressing programming errors and soft errors that affect weights. Their first approach relied on dataset-derived input, while the second optimized output deviation between fault-free and fault-prone models. However, the objective did not explicitly enforce class label changes, leading to limited fault coverage (e.g., 50% on CIFAR-10 in certain configurations).

Other works, such as [54], optimized cross-entropy objectives to induce divergence under weight perturbations. Although effective for specific fault scenarios, these methods often focus on individual faults and generate relatively small sets of patterns without explicitly addressing compactness or storage constraints.

A common characteristic of gradient-based methods is their reliance on internal gradients and model parameters, effectively assuming white-box or gray-box access. Furthermore, many approaches measure fault detection through output deviation (e.g., logit differences), which can be sensitive to runtime variations such as voltage fluctuations and may not reliably indicate functional errors. In sealed deployment scenarios where only predicted class labels are observable, deviation-based metrics may be impractical.

3.0.5. Optimization and Evolutionary Search-Based Methods

To overcome limitations of purely gradient-driven techniques, optimization-based and evolutionary methods have been proposed to search the high-dimensional input space more flexibly.

Chaudhuri et al. [78] employed Bayesian optimization to generate test inputs for systolic array accelerators. The objective maximized the cross-entropy loss between faulty and fault-free outputs, averaged over multiple fault instances to obtain generalized patterns.

Turco et al. [86] applied Evolutionary Algorithmss (EAs) to evolve input images capable of activating permanent hard-to-detect faults in convolutional networks. Their fitness functions evaluated either full output-vector differences or predicted class changes under injected bit-flip faults in floating-point weights.

These methods demonstrate the ability to explore non-convex and non-differentiable search spaces. However, they typically incur high computational cost and often produce large, uncompressed test sets. Compactness and storage efficiency are not primary design objectives, limiting applicability in in-field testing scenarios.

3.0.6. Generative and Hardware-Aware Test Generation

Generation-based methods have also used deep generative models or hardware-aware techniques.

Kundu et al. [76] used conditional Generative Adversarial Networks (GANs) (Conditional Generative Adversarial Networks (cGANs)) to synthesize test images that maximize fault observability under injected permanent bit flips. Ruospo et al. [82] proposed a hardware-aware self-test methodology for GPUs multipliers, constructing structured Image Test Librarys (ITLs) based on internal knowledge of hardware data flow and convolutional layer weights.

Although these approaches can achieve strong fault activation, they often rely on detailed internal knowledge of hardware or model parameters. This assumption may not hold in real-world deployment scenarios where accelerators are delivered as sealed IP blocks.

3.0.7. Black-Box Functional Testing

In realistic deployment scenarios, internal model parameters, gradients, and intermediate activations are often inaccessible due to IPs protection, abstraction layers, or secure execution environments. This constraint motivates black-box functional testing methodologies that rely solely on observable input–output behavior.

Existing black-box approaches typically rely on querying the model with selected or mutated inputs and observing prediction changes without access to gradients or internal representations. However, such methods often struggle to efficiently explore the input space and achieve high fault coverage under strict observability constraints.

3.0.8. Test Compaction and Storage-Efficient Functional Testing

Storage efficiency and test application cost are critical concerns, particularly in in-field or on-device testing scenarios. Many existing generation-based methods produce large test sets, increasing memory requirements and test execution time.

Several works have explored compaction strategies to reduce test-set size after generation. However, most prior approaches follow a two-stage process: first generating a large set of test patterns, and then applying compaction or compression. This separation may lead to inefficiencies, since the generation objectives do not explicitly account for storage constraints. Post hoc compression can also degrade fault coverage if critical high-frequency components or sensitive perturbations are removed.

The classical digital testing literature provides additional inspiration. Techniques such as store-and-generate [5], bit-fixing [7], and bit-flipping [8] demonstrate how compact on-chip generators can transform pseudo-random sequences into deterministic high-coverage test patterns. These approaches emphasize integrating compaction into the generation process itself rather than treating it as a secondary optimization step.

Translating such principles to functional testing for DNNs remains relatively unexplored. Existing DNN focused methods rarely design generation objectives with explicit storage constraints in mind, highlighting the need for frameworks that jointly optimize coverage and compactness.

3.0.9. Functional Built-In Self-Test

BISTs techniques have long been used in digital circuits to enable autonomous, on-chip test generation and response evaluation [64]. Functional BISTs extends this concept beyond structural logic faults by generating input stimuli that activate relevant functional paths at operational speed.

While BISTs has been extensively studied in classical hardware contexts, its adaptation to DNN accelerators presents unique challenges. Unlike deterministic logic circuits, DNNs functionality emerges from learned representations and nonlinear transformations. Generating compact, high-coverage functional patterns under black-box observability and storage constraints requires new abstractions that bridge traditional ATPGs principles and DNNs specific behavior.

To date, no widely adopted, comprehensive BISTs-style framework has been established for functional testing of DNN accelerators. Existing approaches either depend on external test libraries, require internal hardware knowledge, or incur high computational overhead during generation. This gap motivates the exploration of on-demand, storage-efficient functional test generation tailored to deployment constraints.

3.0.10. Unified Fault Modeling and Multi-Fault Optimization

Another important dimension concerns the scope of fault modeling. Many earlier works optimize test patterns for a single injected fault at a time, which may lead to large test sets when extended to broader fault spaces.

Recent approaches attempt to generalize across multiple faults. For example, Bayesian optimization techniques [78] average loss across multiple faulty instances to produce generalized patterns. Similarly, evolutionary methods [86] evaluate fitness in sets of injected faults to improve the detection diversity.

Nevertheless, comprehensive frameworks that explicitly optimize coverage over representative fault lists while controlling test-set size are largely lacking. As fault spaces grow combinatorially (e.g., many weights, bit positions, and layers), scalable abstraction and sampling strategies become essential.

3.0.11. Equivalence Checking for Neural Networks

Beyond hardware-induced faults, functional validation also arises in the context of model transformations, such as pruning, quantization, or architecture modifications. Classical equivalence checking in electronic design automation ensures that two digital circuits implement the same Boolean function [10]. Techniques based on Binary Decision Diagrams (BDDs) [9] and Boolean Satisfiabilitys (SATs) [17] can formally prove functional equivalence for deterministic logic circuits.

However, NNs differ fundamentally from Boolean circuits. Their functionality is specified implicitly through training data rather than explicit truth tables. Consequently, equivalence checking cannot rely on an exhaustive functional comparison. Previous work on NNs equivalence [57], [73] often assumes piecewise-linear activation functions and focuses on verifying equivalence under restricted architectural constraints.

State-of-the-art architectures, including transformer-based models employing softmax and attention mechanisms [36], may not satisfy these assumptions. Moreover, strict formal equivalence may be unnecessary or infeasible in practical settings, where behavioral similarity within the input distribution is sufficient. These observations motivate alternative equivalence-testing methodologies based on the generation of in-distribution counterexamples to reveal functional mismatches.

3.0.12. Summary of Limitations in Existing Work

Although significant progress has been made in DNNs fault detection and functional validation, several limitations persist in existing approaches:

- **Dataset dependence:** Selection- and mutation-based methods rely heavily on existing datasets, limiting exploration of unseen input regions.
- **White-box assumptions:** Many gradient-based and hardware-aware approaches require internal model access, restricting applicability in sealed deployment scenarios.
- **Deviation-based detection:** Reliance on output vector deviation rather than decision-level changes may be impractical under black-box observability and susceptible to runtime noise.
- **Large test sets:** Generation-based methods often produce extensive pattern libraries, increasing storage and application overhead.
- **Two-stage optimization:** Compaction and compression are frequently applied after generation, rather than being integrated into the objective.
- **Limited multi-fault scalability:** Optimizing patterns only per individual fault does not scale efficiently to large fault spaces.
- **Lack of unified frameworks:** Few works simultaneously address coverage, compactness, storage efficiency, and black-box deployability within a single methodology.

These limitations motivate the development of unified functional ATPGs-inspired frameworks for DNN accelerators that explicitly incorporate decision-level observability, fault-list-driven optimization, compact pattern generation, and storage-efficient deployment strategies.

Part II.

Contributions

4. Automatic Test Pattern Generation for DNNs

This chapter presents an ATPG framework for the detection of faults in DNNs. As discussed earlier, DNNs are a fundamental component of modern intelligent systems [69] and are increasingly deployed in safety-critical domains. The reliability of such systems is therefore of paramount importance, particularly since manufacturing defects and runtime faults may impair inference accuracy or lead to critical misclassifications [44], [58]. Unlike conventional digital systems, whose functionality is explicitly specified and can be validated against deterministic reference models [14], DNNs learn their behavior from data, making functional testing inherently more challenging.

To systematically address this challenge, the proposed ATPG framework targets structural faults and detects them through their functional impact at the application level.

The adopted fault model considers SAFs that affect individual neurons in fully connected layers and filters in CNNs, such that the output of the affected computational unit is forced to a constant value, thereby altering the propagation of the internal signal and potentially modifying the final classification decision.

In this context, a test pattern is defined as an input that reveals the presence of one or more injected faults through an observable deviation in the predicted class label compared to fault-free inference.

This definition establishes a direct link between structural hardware defects and misclassification, enabling systematic fault observability analysis.

To mitigate the scalability challenges associated with large fault spaces and extensive test sets, a heuristic test compaction strategy is incorporated. In addition, a clustering-based reduction technique using k-means is applied to identify representative test patterns while maintaining full fault coverage.

The remainder of this chapter is structured as follows. Section 4.1 describes the adopted fault model and presents the proposed ATPG methodology in detail. Section 4.2 reports and analyzes the experimental results. Finally, the chapter concludes with a summary of the findings.

4.1. Fault Modeling and ATPG Framework

This section presents the adopted fault modeling strategy and the proposed ATPG framework, including test generation and compaction mechanisms. The objective is to establish a systematic methodology that links structural faults within a DNN to observable functional deviations at the output level.

4.1.1. Fault Modeling

SAFs are widely considered in the testing of digital circuits at the logic level [16]. In conventional logic testing, such faults abstract physical defects by assuming that a signal line is permanently fixed to a logical value. However, translating this concept into DNNs requires careful abstraction due to their computational and structural differences.

In this work, faults occurring in various internal components of a neuron, including synaptic weight values, Multiply-Accumulate (MAC) operations, or activation functions, are abstracted by modeling faults at the output of the neuron. The rationale behind this abstraction is that internal faults can only be detected and propagated if they ultimately affect the output value of the neuron. This is conceptually analogous to the pin fault model in digital circuit testing, where only faults observable at the output are considered relevant for functional testing.

Consequently, a neuron $n_j^l := \text{Neuron}_j^l(\mathbf{x})$ in an arbitrary layer l is declared to be stuck at a deterministic value v if

$$n_j^l := \text{Neuron}_j^l(\mathbf{x}) = v \quad \forall \mathbf{x} \in \mathbb{X},$$

that is, if the neuron outputs the constant value v independently of the input $\mathbf{x}$. For simplicity, the shorthand notation n_j is used in the following when the layer index l is clear from the context or not required.

The choice of the stuck value v depends on the activation function $\phi(\cdot)$ and its output range. Since internal faults may lead to abnormal neuron behavior, representative signature values from the activation range are selected. For example, in the case of a Tanh activation, extreme values such as $v = -1$ or $v = 1$ correspond to saturated neuron behavior. In addition, intermediate values such as $v = 0$ may also represent faulty behavior depending on the fault mechanism under consideration.

Beyond classical fully connected layers, SAFs are also extended to filters in CNNs. The computation of a convolutional filter can be interpreted as that of a neuron with shared weights. Consequently, fault abstraction is applied to the entire output channel, forcing it to the constant value v determined according to the output range of the activation function. An illustration of SAFs affecting neurons and convolutional filters is shown in Figure 4.1.

As DNNs models increase in size and incorporate regularization techniques such as dropout or weight decay, they may exhibit resilience to individual neuron faults [51]. In such cases, single-fault analysis may not adequately capture functional degradation. Therefore, multiple SAFs applied to a set of neurons, e.g., $\{n_2, n_5, \cdots\}$, are additionally considered. These faults are ranked according to their impact on network accuracy, enabling prioritization of functionally critical fault combinations.

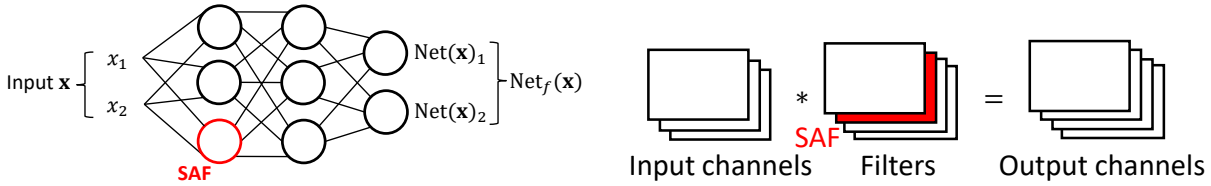

Figure 4.1.: Stuck-at-faults (SAF) in neurons (left) and convolutional filters in CNNs (right).

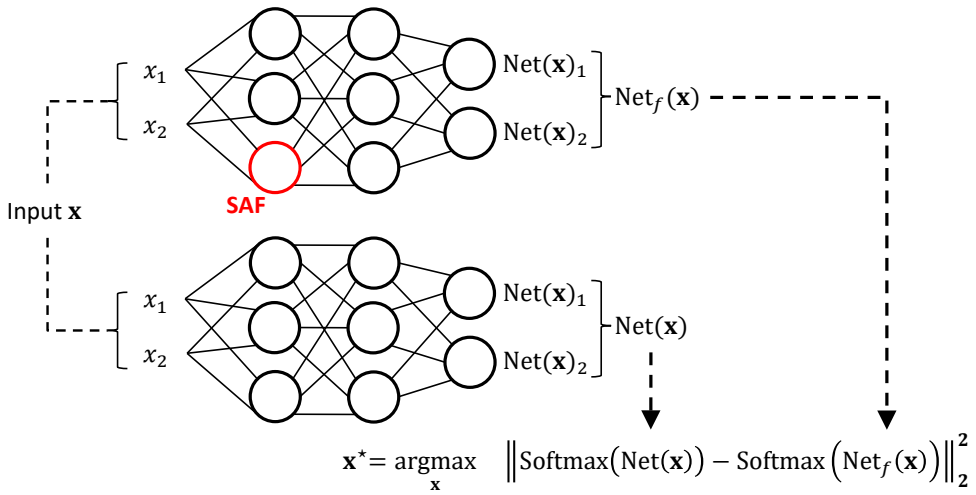

Figure 4.2.: Accuracy degradation after single fault at different layers.

4.1.2. Automatic Test Pattern Generation

For functional testing, it is assumed that internal faults cannot be directly observed. Instead, only input vectors $\mathbf{x}$ can be applied to the network while observing the corresponding outputs $\mathrm{Net}(\mathbf{x})$. A fault is considered detectable for a given input $\mathbf{x}$ if the predicted class label—defined as the index of the maximum output value—differs between the fault-free network $\mathrm{Net}(\mathbf{x})$ and the faulty network $\mathrm{Net}_f(\mathbf{x})$.

The objective of test generation is therefore to identify inputs that maximize the deviation between the fault-free and faulty network outputs such that the classification decision changes. To achieve this, the deviation between the Softmax output distributions of $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}_f(\mathbf{x})$ is maximized. The Softmax layer is particularly relevant, since it couples all output neurons through the normalization constraint that the probabilities sum to one. Increasing the probability of one class necessarily decreases the probability of others, thereby amplifying the sensitivity of the classification to faults.

Various distance measures may be used to quantify this deviation, such as cross entropy [54]. In this work, the squared Euclidean norm of the difference vector is selected due to its symmetry and computational simplicity. The optimal test pattern $\mathbf{x}^\star$ is therefore obtained by solving the following:

$$\mathbf{x}^\star = \underset{\mathbf{x}\in\mathbb{X}}{\operatorname{argmax}} \left\| \mathrm{Softmax}\left(\mathrm{Net}(\mathbf{x})\right) - \mathrm{Softmax}\left(\mathrm{Net}_f(\mathbf{x})\right) \right\|_2^2, \tag{4.1}$$

where $\mathbb{X}$ denotes the feasible input domain. For image classification tasks, this domain typically restricts pixel values to the interval $[0, 255]$. An illustration of this objective function is provided in Figure 4.2.

To generate a test pattern for a specific fault, the corresponding faulty network $\mathrm{Net}_f(\mathbf{x})$ is first constructed. Equation (4.1) is then solved using an appropriate optimization algorithm. Given that $\mathbf{x}$ is often high-dimensional and the objective is differentiable with respect to $\mathbf{x}$, gradient-based optimization methods such as gradient descent or Adam [28], as explained in Section 2.1.4, are well suited.

These methods iteratively update the solution according to:

$$\mathbf{x}^{(t+1)} \leftarrow \mathrm{Proj}_{\mathbb{X}}\left[\mathbf{x}^{(t)} + \alpha^{(t)} \cdot \Delta\mathbf{x}^{(t)}\right],$$

where $\alpha^{(t)}$ denotes the step size and $\mathrm{Proj}_{\mathbb{X}}[\cdot]$ ensures feasibility by projecting the iterate onto the valid input domain. For image inputs, this projection corresponds to clipping pixel values to the allowed range.

The initial solution $\mathbf{x}^{(0)}$ may be selected from the training dataset or randomly generated from $\mathbb{X}$, provided the gradient is non-zero (which may not hold for certain activation functions such as ReLU). Optimization can be restarted multiple times to improve the quality of the solution. Termination criteria include convergence of iterations, stagnation of the objective improvement, or reaching a predefined iteration limit.

Unlike adversarial example generation, the generated test pattern $\mathbf{x}^\star$ is not required to resemble a natural input. Its sole purpose is to reveal functional deviations caused by structural faults. This relaxed constraint increases flexibility in optimization and facilitates subsequent compaction.

4.1.3. Test Pattern Compaction

The ATPG procedure generates one test pattern per targeted fault. For large networks containing numerous neurons and filters, the resulting test set may become excessively large. Therefore, compaction strategies are necessary to reduce the test application time and storage requirements without sacrificing fault coverage.

Test compaction has been extensively studied in combinational circuit testing [6], [12], [19], [71]. Inspired by these approaches, two compaction mechanisms are considered.

The first approach is a heuristic static compaction algorithm that removes redundant test patterns. The algorithm begins by computing the number of faults detected by each test pattern. Patterns detecting

the maximum number of faults are selected first, and redundant detections are iteratively eliminated. After each iteration, the fault coverage is recalculated and the procedure continues until all faults are covered. The resulting compacted pattern indices are stored. The pseudo-code is provided in Algorithm 1. The computational complexity is dominated by the evaluation of fault-pattern pairs, resulting in $O(N^2)$ complexity for N patterns.

Algorithm 1 The heuristic test pattern compaction algorithm.

```
1:  function Test_Compaction(patterns, faults, fault_count)
2:      i, j ← 0
3:      Generated_test_index_list ← empty_list
4:      for i in patterns do
5:          max_fault_count ← max(fault_count)
6:          image_index ← index[max_fault_count]
7:          Generated_test_index_list.append(image_index)
8:          for j in faults do
9:              if cell(Generated_test_index, j ) == cell(i, j) then
10:                 remove_cell(i, j)
11:             end if
12:         end for
13:         recalculate_fault_counts_in_all_patterns()
14:         if max(fault_count) == 0 then
15:             break;
16:         end if
17:     end for
18:     return (Generated_test_index_list)
19: end function
```

Alternatively, cluster-based compaction using K-means is applied. Since test patterns detecting similar faults are likely to be close in the input space, clustering can identify groups of redundant patterns. The classical K-means algorithm is employed and may be executed multiple times with varying cluster counts K. The best cluster configuration is selected using evaluation metrics such as the silhouette score [4].

4.2. Experimental Evaluation and Analysis

This section describes the experimental setup used to evaluate the proposed ATPG framework and provides a comprehensive analysis of the obtained results. The generated test patterns are compared against randomly generated inputs and adversarial examples, which represent a state-of-the-art approach to test generation for NNs.

4.2.1. Datasets and Experimental Setup

The evaluation was conducted using the MNIST and CIFAR-10 datasets [24], [25].

All networks were trained using various activation functions and layer configurations to analyze robustness under different architectural settings. The SAFs described in Section 4.1.1 were injected into the trained models. For MLP and the fully connected layers of CNNs, faults were injected at the neuron output. For CNNs, SAFs were additionally applied to convolutional filters, resulting in faulty output channels. All implementations were realized in *PyTorch*.

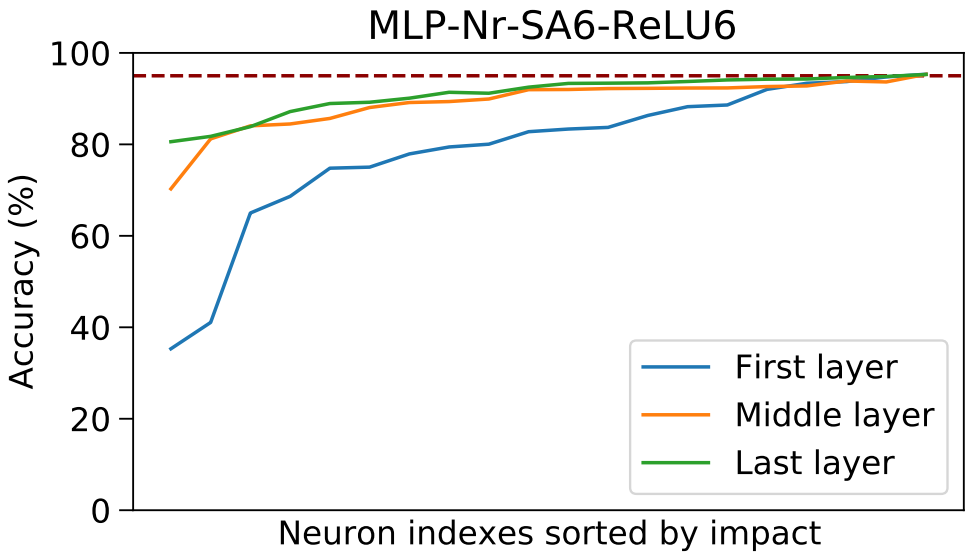

Figure 4.3.: Accuracy degradation after single fault at different layers.

The learning rate for training was determined through a logarithmic grid search ranging from 10^{-7} to 1. For MLP experiments, a model consisting of three hidden layers with 20 neurons each was used. Training was carried out for ten epochs with a batch size of 32.

For CNN experiments, an architecture composed of four convolutional layers followed by three fully connected layers was employed. The models were trained for 20 epochs using a batch size of 32 and exclusively utilized ReLU and ReLU6 activation functions.

4.2.2. Results and Discussion

Following the fault modeling described in Section 4.1.1, functional faults were injected into MLP and the fully connected layers of CNNs. The resulting degradation in classification accuracy was measured. Neurons n_j causing substantial reductions in accuracy were subsequently ranked and prioritized for test pattern generation.

In experiment MLP-Nr-SA6-ReLU6 (see Figure 4.3), the impact of a single SAF on neurons across different network layers was analyzed. Prior to fault injection, the classification accuracy reached 95%. The results show a clear degradation pattern depending on the layer location of the fault. Faults injected into early layers caused more severe accuracy drops compared to faults in later layers. Similar trends were observed across experiments with different activation functions and stuck-at values. This indicates that networks exhibit increasing robustness to single SAFs in deeper layers.

Furthermore, differences in robustness were observed across activation functions. Sigmoid activation demonstrated lower accuracy degradation compared to ReLU and ReLU6. This behavior may be attributed to the saturation properties of Sigmoid, which can partially mask the effects of SAFs. The chosen stuck-at value also influences the observed degradation. For example, SA0 faults often resulted in smaller accuracy drops, particularly for ReLU networks, as ReLU outputs zero for all negative inputs, making SA0 less disruptive in certain regions of the input space.

Fault injections were also applied to the last hidden layer of the fully connected layers in CNN models. Similar to MLP, earlier CNNs layers were more severely affected by SAFs compared to later layers. However, CNNs architectures overall exhibited greater robustness to single SAFs than MLP. Consequently, the impact of multiple simultaneous SAFs was further investigated. As expected, increasing the number of injected faults led to progressively larger reductions in classification accuracy.

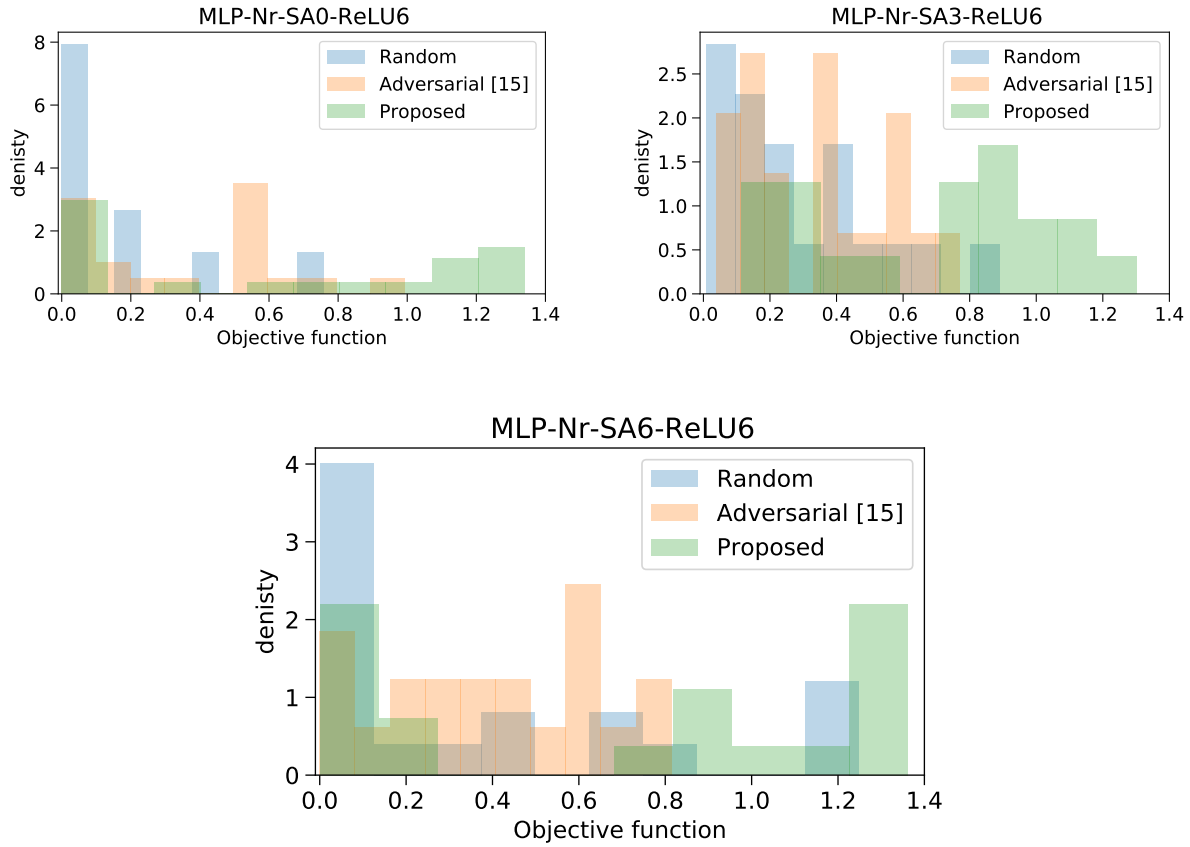

Figure 4.4.: Output deviation for generated patterns at different stuck-at values: SA0 (top-left), SA3 (top-right), and SA6 (bottom).

Additionally, SAFs were injected into convolutional filters and ranked according to severity. As observed in MLP experiments, injecting multiple faults resulted in more pronounced accuracy degradation.

After completion of the fault injection analysis, test patterns $\mathbf{x}^\star$ were generated for each injected neuron n_j or channel. Each test pattern was obtained by solving Eq. (4.1) using the Adam optimizer for 500 iterations. A warm-start procedure was applied by running the optimizer for five iterations with a small step size to initialize moment estimates.

The evaluation measured the percentage of misclassified class labels after fault injection using the generated test patterns. These results were compared with those obtained using random test images and adversarial inputs [45]. The misclassification percentages are reported in Table 4.1. The results demonstrate that the proposed ATPG approach consistently achieves the highest misclassification rates, indicating superior fault coverage compared to both random and adversarial inputs. In numerous experiments, adversarial inputs failed to produce any misclassification, whereas the proposed ATPG patterns achieved up to 100% misclassification.

In experiments such as MLP-Nr-SA0-ReLU and CNN-Nr-SA0-ReLU, the misclassification percentage was relatively modest, although still higher than that of adversarial and random patterns. This behavior can be explained by the properties of the ReLU activation function, which produces zero outputs for a significant portion of its input domain. However, when multiple SA0 faults were injected, as in CNN-Nr-SA0(10%)-ReLU, the misclassification rate increased to 100%.

To further analyze the results, the deviation between fault-free and faulty outputs was calculated for all generated test patterns. These deviations were compared to those obtained from random inputs and

Table 4.1.: Experimental results

Experimental setup			Test pattern approaches						Compaction ratio (~)	
			Random		Adversarial [45]		Proposed			
Model	Activation	Stuck at faults	Mean ± Standard Deviation	Incorrect labels (%)	Mean ± Standard Deviation	Incorrect labels (%)	Mean ± Standard Deviation	Incorrect labels (%)	Heuristic	Clustering
MLP	Sigmoid	SA0	0.16 ± 0.25	15	0.24 ± 0.19	20	0.85 ± 0.28	95	6.3x	4.7x
		SA0.5	0.19 ± 0.33	15	0.24 ± 0.21	15	0.86 ± 0.33	95	6.3x	9.5x
		SA1	0.22 ± 0.39	20	0.24 ± 0.26	15	0.85 ± 0.40	95	6.3x	3.1x
		SA0(10%)	0.16 ± 0.22	20	0.35 ± 0.33	20	1.01 ± 0.21	90	3.0x	3.3x
	Tanh	SA-1	0.18 ± 0.30	20	0.08 ± 0.17	0	0.57 ± 0.46	75	3.7x	2.5x
		SA0	0.17 ± 0.20	15	0.02 ± 0.03	0	0.59 ± 0.28	85	5.6x	3.4x
		SA1	0.17 ± 0.32	10	0.02 ± 0.03	0	0.53 ± 0.42	65	4.3x	6.5x
		SA1(10%)	0.26 ± 0.33	20	0.03 ± 0.06	0	0.87 ± 0.46	80	2.0x	1.8x
	ReLU	SA0	0.18 ± 0.37	10	0.21 ± 0.30	15	0.78 ± 0.70	55	3.6x	3.6x
		SA6	0.30 ± 0.40	15	0.45 ± 0.36	35	1.27 ± 0.29	100	6.6x	2.8x
		SA0(10%)	0.47 ± 0.56	30	0.46 ± 0.41	40	1.41 ± 0.00	100	3.3x	2.6x
	ReLU6	SA0	0.16 ± 0.23	10	0.36 ± 0.30	50	0.6 ± 0.54	55	3.6x	2.7x
		SA3	0.28 ± 0.23	20	0.34 ± 0.21	35	0.69 ± 0.37	75	5.0x	2.6x
		SA6	0.37 ± 0.43	35	0.4 ± 0.25	35	0.69 ± 0.55	60	4.0x	4.0x
		SA6(10%)	0.39 ± 0.48	30	0.55 ± 0.27	60	0.82 ± 0.52	70	4.5x	4.5x
CNN FC Layers	ReLU	SA0	0.04 ± 0.10	5	0.09 ± 0.25	5	0.09 ± 0.25	45	1.8x	7.0x
		SA6	0.08 ± 0.14	10	0.14 ± 0.20	5	0.77 ± 0.49	95	4.7x	10.0x
		SA0 (10%)	0.11 ± 0.29	10	0.13 ± 0.37	10	0.96 ± 0.63	70	2.6x	2.6x
	ReLU6	SA0	0.13 ± 0.17	10	0.14 ± 0.14	45	0.7 ± 0.34	85	3.4x	5.3x
		SA3	0.06 ± 0.09	0	0.13 ± 0.09	25	0.39 ± 0.16	100	6.6x	2.2x
		SA6	0.01 ± 0.03	0	0.25 ± 0.21	20	0.68 ± 0.36	80	3.2x	10.0x
		SA6 (10%)	0.29 ± 0.38	20	0.34 ± 0.24	20	1.04 ± 0.21	100	5.3x	2.2x
CNN Conv Layers	ReLU	SA0	0.08 ± 0.15	5	0.04 ± 0.16	3	0.69 ± 0.69	50	1.9x	10.3x
		SA6	0.85 ± 0.59	66	1.17 ± 0.37	88	1.41 ± 0.00	100	15.5x	10.3x
		SA6 (~6%)	1.20 ± 0.33	94	0.92 ± 0.60	62	1.41 ± 0.00	100	8.0x	10.0x
	ReLU6	SA0	0.06 ± 0.08	5	0.12 ± 0.15	11	1.24 ± 0.22	100	2.5x	2.5x
		SA3	0.97 ± 0.37	80	0.99 ± 0.38	81	1.41 ± 0.02	100	5.0x	2.5x
		SA6	0.99 ± 0.25	91	0.95 ± 0.23	98	1.41 ± 0.01	100	5.0x	2.5x
		SA0 (~6%)	0.26 ± 0.41	19	0.20 ± 0.27	19	1.23 ± 0.46	88	2.6x	5.3x
Average			0.25	24	0.21	19	2.84	83	4.6x	4.8x

Table 4.2.: Runtime for compaction methods and the effect of combining them together.

Experimental setup		Runtime (~sec)		Compaction ratio (~)		
Model	Activation	Heuristic	Clustering	Step 1. Heuristic	Step 2. Clustering (after Heuristic)	Combined compaction results
MLP	Sigmoid	0.0015	0.708	6.3x	2.0x	12.6x
	Tanh	0.0019	0.493	4.5x	2.0x	9.0x
	ReLU	0.0014	0.461	5.1x	3.0x	15.3x
	ReLU6	0.0015	0.336	4.2x	1.5x	6.3x
CNN	ReLU	0.0018	0.988	6.8x	2.5x	17.0x
	ReLU6	0.0016	0.975	5.6x	1.4x	7.8x
Average		0.0016	0.66	5.4x	2.0x	11.3x

adversarial test images generated via FGSM (see Eq. (3.1)). Figure 4.4 presents histogram representations of the output deviations. A rightward shift in the histogram corresponds to larger output differences. Across all experiments, the proposed ATPG approach achieved the largest deviations compared to random and adversarial inputs.

The mean and standard deviation of output deviations were also computed and are reported in Table 4.1. The ATPG approach consistently achieved the highest output variation, further confirming its effectiveness in exposing functional faults.

Furthermore, two compaction strategies were applied as described in Section 4.1.3. The first is a heuristic redundancy elimination algorithm, and the second is a clustering-based method that groups similar test patterns. The compaction ratio—defined as the ratio between the size of the original and compacted test sets—is reported in Table 4.1. Both approaches significantly reduced the test set size. On average, clustering achieved a slightly higher compaction ratio than the heuristic method.

Runtime analysis was also conducted. Since the computational cost is dominated by gradient evaluations, and the proposed ATPG uses 500 optimization iterations compared to a single gradient step in FGSM, the runtime is expected to scale proportionally with the number of iterations. Runtime measurements for the compaction approaches are summarized in Table 4.2, where mean values across activation functions and stuck-at settings are reported. The heuristic method consistently exhibited lower runtime compared to clustering, revealing a trade-off between compaction efficiency and computational cost.

Finally, the combination of both compaction strategies was investigated. The heuristic method was first applied to eliminate redundant patterns, followed by clustering to further group similar inputs. The total compaction ratio was computed relative to the original test set size. As shown in Table 4.2, applying clustering after heuristic compaction further reduced the final test set size, demonstrating the complementary nature of both techniques.

4.3. Chapter Summary

This chapter presented an ATPG framework for detecting functional neuron faults in DNNs. The approach formulates test generation as an optimization problem that maximizes the deviation between the Softmax outputs of fault-free and faulty networks, thereby enforcing observable misclassification in the presence of structural faults.

Experimental results show that the generated patterns achieve higher misclassification rates and larger output deviations than adversarial and random inputs, demonstrating the effectiveness of the proposed ATPG strategy.

Two static test compaction techniques were also introduced: heuristic redundancy elimination and clustering-based reduction using K-means. Both methods significantly reduce the test set size while maintaining full fault coverage, highlighting a trade-off between compaction efficiency and runtime. Combining both approaches further improves compaction.

5. Compact Functional Testing Under Multiple Faults

5.1. Chapter Overview

This chapter presents a compact functional testing framework for detecting multiple simultaneous faults in DNNs implemented on hardware accelerators. Building upon the single-fault analysis of the previous chapter, the focus is extended to fault patterns, i.e., instances of multiple faults, and their efficient detection using compact test sets.

Traditional digital testing methodologies typically rely on the single-fault assumption. Although this abstraction is well established for conventional logic circuits, it is often insufficient for DNN accelerators. As demonstrated in the previous chapter, individual faults may have a limited impact due to the inherent redundancy and resilience of NNs architectures. In contrast, multiple simultaneous faults can significantly degrade inference accuracy. From a technological perspective, the large scale of modern accelerators and the variability of emerging technologies, particularly memristive devices, increase the likelihood of concurrent faults.

Testing under multiple-fault scenarios introduces substantial challenges. First, the number of possible fault combinations grows exponentially with the number of fault sites, making exhaustive evaluation difficult. Second, evaluating large datasets to expose such faults is computationally expensive and time-consuming [56]. Therefore, efficient and compact test pattern generation capable of detecting representative fault patterns is essential.

In this chapter, multiple faults are modeled at the level of synaptic weight representations, specifically considering bit-level faults that are representative of common hardware defects in digital- and memory-based accelerator implementations. Fault detection is performed through functional misclassification, thereby establishing a link between structural bit-level perturbations and observable deviations in network predictions.

To address the combinatorial explosion of fault combinations, a structured fault list is first constructed to represent targeted fault patterns. Based on this list, a compact set of test patterns is generated to achieve high detection rates on the targeted faults. Furthermore, the generalization capability of the generated patterns is evaluated by applying them to non-targeted fault instances not explicitly included in the fault list. In addition to fault detection, compaction is integrated into the framework to reduce test set size while maintaining high coverage, thereby improving scalability for larger network architectures.

The remainder of this chapter is organized as follows. Section 5.2 presents the proposed compact test generation framework. Section 5.3 provides the experimental evaluation and analysis. Finally, the chapter concludes with a summary of the main findings and implications.

5.2. Multi-Fault Modeling and Compact Test Generation Framework

This section introduces the adopted multi-fault model and presents the compact test pattern generation strategy designed to efficiently detect fault patterns in DNNs accelerators.

5.2.1. Multi-Fault Modeling

Bit-level fault injection has been widely considered in the evaluation of hardware reliability for NNs. In this chapter, faults are modeled by perturbing the bit-level representation of a randomly selected subset of network parameters (see Section 2.3).

Let p_w denote the percentage of weights affected by faults and p_b denote the percentage of altered bits within each selected weight. A pair (p_w, p_b) therefore defines a specific instance of the fault model considered. Since multiple weights and multiple bit positions are altered simultaneously, this abstraction naturally corresponds to a multiple-fault model, in contrast to traditional single-fault assumptions.

This modeling approach reflects realistic hardware defect scenarios, particularly in large-scale accelerators and memory-based implementations, where correlated and concurrent bit-level faults may occur.

5.2.2. Compact Test Pattern Generation

In many practical deployment scenarios, such as memristor-based crossbars or dedicated neural accelerators, the circuit can be regarded as a black box. That is, only inputs $\mathbf{x}$ can be applied, and the corresponding classification labels are observed at the output. Consequently, effective test patterns must reveal faults through observable label mismatches between the fault-free network Net and a faulty instance Net_f.

To encourage divergent classification results between $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}_f(\mathbf{x})$ for a given fault f, the following objective function is adopted:

$$o(\mathbf{x}, f) := \left\| \mathrm{Softmax}\left(\mathrm{Net}(\mathbf{x})\right) - \mathrm{Softmax}\left(\mathrm{Net}_f(\mathbf{x})\right) \right\|_2^2 .$$

Maximizing $o(\mathbf{x}, f)$ with respect to $\mathbf{x}$ yields an input pattern for which the Softmax outputs of fault-free and faulty networks differ significantly. Since the Softmax operation couples all output entries (see Eq. (2.4)), reducing the highest probability inherently increases competing class probabilities. Therefore, maximizing the deviation in Softmax distributions strongly promotes classification mismatches.

For practical applicability, test patterns should not only detect a single fault instance but also generalize to similar fault variations. To facilitate this, a fault list $\mathcal{F}_{(p_w,p_b)} = \{f_n \mid n = 1, \ldots, N\}$ is constructed for a given (p_w, p_b). Each f_n represents a randomly generated multi-fault instance associated with a faulty network Net_{f_n}.

Although the fault list does not exhaustively represent all possible fault combinations, it serves as a representative subset to approximate overall fault coverage. To avoid excessive storage overhead, individual fault instances are not stored explicitly. Instead, each f_n is associated with a random seed n, allowing deterministic regeneration of the fault pattern when required.

The objective is to generate test patterns capable of detecting multiple fault instances within the fault list. To achieve high coverage with minimal patterns, the following optimization problem is formulated:

$$\mathbf{x}^\star = \underset{\mathbf{x}\in\mathbb{X}}{\mathrm{argmax}} \min_{f\in\mathcal{F}} o(\mathbf{x}, f). \tag{5.1}$$

Rather than maximizing $o(\mathbf{x}, f)$ for a single fault, this formulation maximizes the minimum deviation between all faults in $\mathcal{F}$. In other words, $\mathbf{x}^\star$ maximizes the lower bound of $o(\mathbf{x}, f)$ over the entire fault list. As a result, the generated pattern is encouraged to induce misclassification for many fault instances, thereby achieving high overall fault coverage.

Since a single test pattern is unlikely to detect all fault instances, a sequential extraction procedure is employed. After computing the first pattern $\mathbf{x}_1^\star$, all detected faults are removed from the fault list. The optimization in Eq. (5.1) is then repeated on the reduced fault set to obtain $\mathbf{x}_2^\star$, and the process continues until the fault list becomes empty. The resulting test set $\mathcal{X} = \{\mathbf{x}_k^\star\}$ achieves complete coverage of the target fault list.

The optimization problem in Eq. (5.1) can be solved using gradient-based algorithms such as gradient descent or Adam [27]. Gradients are computed via automatic differentiation [49]. Step sizes may be chosen constant or follow a decay schedule.

Throughout optimization, input constraints $\mathbf{x} \in \mathbb{X}$ must be enforced. For image-based tasks, this usually requires the pixel values to remain within $[0, 255]$. Feasibility is maintained through element-wise projection, commonly referred to as clipping.

The $\min_{f \in \mathcal{F}}$ operation in Eq. (5.1) does not impede optimization. Since $\mathcal{F}$ is finite, the minimum can be computed by evaluating $o(\mathbf{x}, f)$ for all $f \in \mathcal{F}$ and selecting the smallest value. Furthermore, the objective remains differentiable with respect to $\mathbf{x}$ (almost everywhere), as each $o(\mathbf{x}, f)$ is differentiable.

In summary, for a given (p_w, p_b), a representative fault list $\mathcal{F}_{(p_w,p_b)}$ is constructed. Test patterns are sequentially extracted to maximize collective coverage over $\mathcal{F}$. Detected faults are iteratively removed until full coverage is achieved. The resulting compact test set can subsequently be applied to detect additional, non-targeted fault instances not explicitly contained in the fault list.

By analogy, the fault list may be interpreted as a training set for designing fault-revealing patterns. Although an exhaustive enumeration of all multiple-fault combinations is infeasible, this approach enables high fault coverage without explicit modeling of the entire combinatorial fault space. The complete procedure is summarized in Algorithm 2.

Algorithm 2 The test pattern generation algorithm for a given network Net. The set of found test patterns is stored in $\mathcal{X}$.

```
# Set design parameters
Choose p_w, p_b, N
# Generate fault list
F ← F_(p_w,p_b) = {f_n | n = 1, ··· , N}
# Define covered/detected fault set for test pattern x
F_c(x) := {f ∈ F | argmax_i Net(x)_i ≠ argmax_i Net_f(x)_i}
# Initialize
k ← 0
X ← ∅
while F ≠ ∅ do
    k ← k + 1
    x*_k ← argmax_{x∈X} min_{f∈F} o(x, f)
    F ← F \ F_c(x*_k)
    X ← X ∪ {x*_k}
end while
```

5.3. Experimental Evaluation and Analysis

The proposed compact test patterns are evaluated on networks trained on two commonly used benchmark datasets under various configurations of p_w (percentage of weights affected by bit-flips) and p_b (percentage of flipped bits per affected weight).

5.3.1. Datasets and Experimental Setup

The experimental evaluation was conducted on MNIST and CIFAR-10 using 32-bit floating-point weight representations implemented in *PyTorch*.

Table 5.1.: The overall experimental results on different models and datasets, p_w: percentage of fault injected weights, p_b: percentage of random bit flips per fault-injected weight, F_P: number of fault patterns. T_P: number of test patterns, T_f: targeted faults. U_f: unseen faults. CR: compaction ratio. $iter$: number of iterations. The random images provide 100% coverage for the targeted faults. One adversarial test image is generated to test for targeted and unseen faults. The number of fault patterns in the non-targeted list is 1000.

Experimental setup				**Test pattern approaches**											
				Random			**Adversarial** [45]			**Proposed**					
Model and Dataset	p_w (%)	p_b (%)	$\#F_P$	$\#T_P$	**Test Coverage (%)**	**Time (~Sec)**	**Test coverage (%)**		**Time (~Sec)**	$\#T_P$	**CR (~)**	**Test Coverage (%)**		**Time (~Sec)**	$\#iter$
					U_f		T_f	U_f				T_f	U_f		
MLP (MNIST) 3 FC layers Size 20	5	100	50	93	97.0	0.4	84.0	71.3	0.03	8	6.2×	100	98.8	88	14
			100	72	98.6	3.4	72.0	71.3	0.11	9	11.1×	100	98.8	120	14
			150	82	94.0	6.7	70.6	71.3	0.31	10	15.0×	100	98.8	189	14
			200	150	98.2	6.6	73.5	71.3	0.02	10	20.0×	100	98.8	251	14
	10	70	50	2250	98.5	15.89	76.0	80.6	0.07	3	16.7×	100	98.9	29	3
			100	1284	98.6	28.57	86.0	82.2	0.08	5	20.0×	100	98.7	59	22
			150	1200	98.5	19.57	86.0	82.2	0.12	5	30.0×	100	98.7	86	22
			200	2229	98.8	22.99	82.5	82.2	0.19	6	33.3×	100	99.0	123	22
	10	50	50	1632	92.3	16.80	64.0	63.3	0.09	5	10.0×	100	96.8	39	22
			100	5064	97.0	31.87	69.0	63.3	0.16	6	16.7×	100	98.0	68	22
			150	1432	96.1	19.47	67.0	63.3	0.25	7	21.4×	100	98.3	115	40
			200	2116	97.1	28.8	65.0	63.3	0.36	9	22.2×	100	98.3	153	40
LeNet-5 (MNIST) 3 Conv + 2 FC layers	5	100	50	119	95.0	15.5	78.0	79.4	0.65	6	8.3×	100	98.1	126	12
			100	56	93.9	24.8	78.0	79.4	1.29	8	12.5×	100	98.0	318	12
			150	65	94.0	26.35	80.6	79.4	1.76	9	16.7×	100	99.8	514	33
			200	60	95.0	32.3	82.0	79.4	2.11	9	22.2×	100	99.8	733	33
	10	70	50	1120	95.0	331.7	62.0	68	2.13	8	6.2×	100	97.3	323	12
			100	1117	95.8	369.8	63.0	63	3.84	13	7.7×	100	99.1	604	25
			150	5625	95.2	1104.0	60.3	63	6.27	13	11.5×	100	99.0	861	25
			200	6344	96.1	2053.9	54.4	54.9	10.37	15	13.3×	100	98.9	1447	48
	10	50	50	3885	86.8	795.4	34.0	41.7	3.59	10	5.0 ×	100	97.1	781	44
			100	1858	97.0	928.8	34.0	42	3.70	9	11.1×	100	96.3	1927	31
			150	2087	93.0	1120.4	38.6	38.3	4.63	14	10.7×	100	99.2	82	33
			200	6155	96.0	2113.6	42.5	38.3	2.50	9	22.2×	100	99.3	751	30
ConvNet-7 (CIFAR-10) 4 Conv + 3 FC layers	5	100	50	353	98.0	392.0	80.0	86.6	3.65	2	25.0×	100	99.9	327	6
			100	437	99.0	1090.2	87.0	86.6	5.65	3	33.3×	100	100.0	832	6
			150	602	94.6	2256.6	82.0	80.7	4.70	3	50.0×	100	100.0	1225	6
			200	1977	98.0	3737.3	87.5	86.5	5.10	4	50.0×	100	100.0	1748	6
	10	70	50	3752	98.6	1780.3	72.3	74.9	4.02	10	5.5×	100	99.3	709	10
			100	6206	98.7	2133.3	77.5	78.6	5.46	10	10.0×	100	99.8	1213	10
			150	2387	99.0	1368.8	66.8	70.9	5.71	10	15.0×	100	99.8	1804	10
			200	6598	97.3	2466.2	74.8	75.0	5.39	11	18.1×	100	99.9	2497	13
	10	50	50	2624	94.7	1465.3	78.3	79.1	4.21	7	7.1×	100	98.8	1367	8
			100	3699	97.7	1886.7	68.3	71.9	5.20	11	9.0×	100	98.9	2639	11
			150	4722	98.2	2298.5	87.2	87.1	4.92	19	7.9×	100	99.4	6548	38
			200	6623	98.4	2597.0	79.3	81.4	4.33	21	9.5×	100	99.5	5521	24
VGG-19 (CIFAR-10) 16 Conv + 3 FC	0.01	100	50	5152	98.9	9074	88.0	89.1	5123	2	25.0×	100	99.5	12,010	3
		100	100	6181	99.0	12,187	87.0	84.0	12,242	3	33.3×	100	99.0	26,950	4
		70		6226	92.0	13,710	88.8	89.5	13,523	5	20.0×	100	99.9	30,180	11
	0.1	100	50	4687	88.0	10,519	85.4	80.2	5151	3	16.6×	100	99.9	15,392	5
			100	7762	96.8	14,628	86.0	79.0	10,800	3	33.3×	100	99.2	28,219	3
		70	50	8804	98.0	10,183	86.2	81.3	6002	4	12.5×	100	99.0	14,281	6
			100	8465	95.2	13,987	87.5	81.0	13,111	4	25.0×	100	99.9	29,323	4

For MNIST, two models were considered: (i) an MLP consisting of three fully connected layers of size 20, and (ii) LeNet-5, comprising seven layers, including three convolutional layers, two subsampling layers, and two fully connected layers.

For CIFAR-10, two architectures were evaluated: (i) a custom ConvNet-7 with four convolutional layers and three fully connected layers, and (ii) VGG-19 [30], a deep CNN composed of 19 layers (16

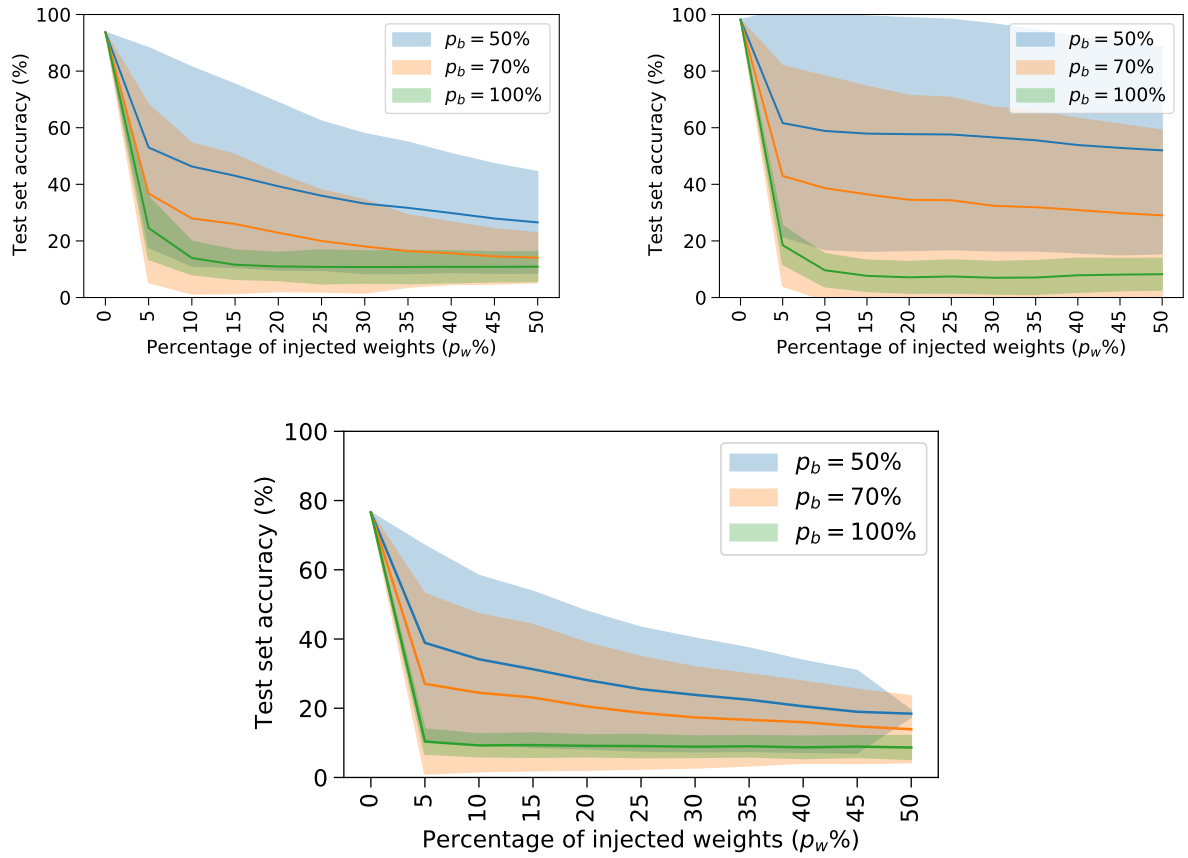

Figure 5.1.: Test set accuracy degradation on different models after bit level fault injection: MLP (top-left), LeNet (top-right), and ConvNet (bottom).

convolutional layers, 3 fully connected layers, 5 max-pooling layers, and one Softmax layer), which serves to demonstrate the scalability of the proposed method on larger networks.

All networks employ ReLU activation functions. The learning rate was determined via logarithmic grid search over the interval $[10^{-7}, 1]$.

5.3.2. Results and Discussion

To analyze the general impact of the considered bit-level fault model (Section 5.2.1), the aforementioned networks were injected with 100 randomly generated fault patterns for various combinations of (p_w, p_b). The mean and standard deviation of the resulting classification accuracy were computed. The results are shown in Fig. 5.1. Even moderate fault configurations, such as flipping 50% of the bits in only 5% of the weights, already lead to significant degradation in accuracy across all evaluated architectures.

Following this preliminary analysis, compact test pattern sets were generated for each network as described in Section 5.2. Results are reported for different (p_w, p_b) combinations and varying fault list sizes $\{50, 100, 150, 200\}$. The detailed results are provided in Table 5.1.

For all evaluated configurations, full (100%) coverage of the targeted fault list was achieved using only a small number of generated test patterns. The compaction rate, defined as the ratio between the fault list size and the number of generated test patterns, exceeded 5× in most cases. Notably, larger networks such as ConvNet-7 and VGG-19 on CIFAR-10 achieved compaction ratios of up to 50×, demonstrating improved scalability for more complex models.

The number of optimization iterations required to achieve complete coverage is also reported in Table 5.1. LeNet-5 required more iterations compared to the other models, potentially due to the difficulty of fault scenarios involving 10% of weights with 50% or 70% bit flips. Execution time for test pattern generation was additionally measured. As expected, smaller models such as the MLP required less time compared to LeNet-5, ConvNet-7, and VGG-19. Since the dominant computational cost arises from gradient evaluations in the optimization routine, larger models incur higher generation times due to increased computational complexity.

Furthermore, faults involving higher bit-flip percentages (e.g., 100%) were detected more rapidly, indicating that faults causing stronger accuracy degradation are easier to expose through optimization.

The observed increase in runtime with growing model size is primarily attributed to the cost of evaluating the objective function during gradient computation. However, an important scalability advantage of the proposed framework lies in the fact that the number of required test generation runs depends only on the size of the fault list and not directly on the number of network parameters. Thus, although individual optimization runs become more computationally intensive for larger networks, the number of runs remains constant. This property ensures scalability in terms of test pattern count and overall testing overhead.

To further assess effectiveness, the proposed approach was compared with random test images and adversarial examples [26]. As shown in Table 5.1, a substantially larger number of random test images was required to achieve full coverage of the fault list. In challenging configurations (e.g., 10% of weights with 50% or 70% bit flips), more than 6000 random images were required. Although individual evaluations of random inputs are computationally inexpensive, the large number of required patterns significantly increases test storage and application time.

Adversarial test images were also generated and evaluated. While adversarial generation incurs relatively low computational cost per image, coverage of the targeted fault list remained limited (e.g., only 34% for LeNet-5), demonstrating that adversarial perturbations are not well aligned with the objective of multi-fault detection.

Finally, the generalization capability of the generated test patterns was evaluated by applying them to 1000 additional fault instances not included in the original fault list. This evaluation provides an estimate of expected field fault coverage. Consistent with previous observations, the proposed test patterns achieved up to 99.9% coverage on these unseen faults while requiring significantly fewer patterns than random or adversarial approaches. This confirms that the compact multi-fault testing strategy effectively generalizes beyond explicitly modeled fault instances.

5.4. Chapter Summary

This chapter presented a compact functional testing framework for detecting multi-fault patterns in DNNs at the bit level of synaptic weight representations. The approach formulates test generation as an optimization problem that maximizes the minimum deviation between the Softmax outputs of fault-free and faulty networks, enabling test patterns that target multiple faults simultaneously and improve overall coverage.

An iterative extraction procedure was introduced to construct a compact test set. In each iteration, a test pattern detects as many remaining fault instances as possible, after which the detected faults are removed from the fault list. This process efficiently reduces the test set size while maintaining full coverage.

Experimental results show that the proposed method consistently outperforms random and adversarial test images. It achieves 100% coverage of targeted fault patterns across different datasets and architectures, with higher compaction ratios for larger networks. Additionally, the generated patterns generalize well to unseen faults, reaching up to 99.9% coverage on non-targeted fault instances.

6. Evolutionary Optimization for Compact Functional Testing

6.1. Chapter Overview

This chapter introduces an evolutionary optimization framework for the generation of compact functional test patterns in DNNs accelerators. In contrast to the gradient-based methods presented in previous chapters, this approach formulates test generation and compaction as a unified black-box optimization problem, enabling the direct minimization of the test set size while maintaining high fault coverage.

An essential component of functional testing is the generation of effective test patterns that expose hardware faults during inference. Previous chapters addressed this problem using gradient-based optimization techniques [55], [80]. Although effective, these approaches typically treat test pattern generation and compaction as separate stages: first generating a potentially large set of patterns, followed by heuristic reduction. This separation limits the ability to directly optimize the size of the resulting test set.

In this chapter, test generation and compaction are integrated into a single optimization framework. The objective is to simultaneously maximize fault detection capability and minimize the number of required test patterns. The considered faults correspond to structural abstractions commonly used to model defects in systolic or memory-centric DNNs accelerators. However, detection remains functional: a fault is considered detected only if it produces an observable output deviation compared to the fault-free model.

Since the size of a test set is inherently non-differentiable, classical gradient-based optimization becomes inadequate. To address this limitation, a black-box evolutionary optimization strategy based on the CMA-ES algorithm is employed. This method enables direct optimization of compact test sets without requiring differentiability of the objective.

The remainder of this chapter is organized as follows. Section 6.1.1 introduces background on evolutionary algorithms and the covariance matrix adaptation evolution strategy. Section 6.1.2 describes the fault model used in this work. Section 6.2 formulates the optimization problem and discusses the integration of generation and compaction. Section 6.3 presents the evolutionary optimization framework based on CMA-ES. Section 6.4 provides the experimental evaluation and comparative analysis. Finally, the chapter concludes with a summary of the key findings.

6.1.1. Evolutionary Algorithms and Covariance Matrix Adaptation Evolution Strategy

EA are black-box optimization techniques inspired by natural selection. They evolve a population of candidate solutions over generations through selection, mutation, and recombination. EA are particularly useful in problems where the objective function is non-differentiable.

A well-known EA for continuous optimization is CMA-ES [13]. In CMA-ES, candidate solutions are sampled from a multivariate normal distribution.

$$\mathbf{x}_k^{(g+1)} \sim \mathcal{N}\left(\mathbf{m}^{(g)}, (\sigma^{(g)})^2 \mathbf{C}^{(g)}\right), \quad k = 1, \ldots, K \tag{6.1}$$

where $\mathbf{m}^{(g)}$ is the mean of the search distribution in generation g, $\sigma^{(g)}$ is the global step-size, $\mathbf{C}^{(g)}$ is the covariance matrix and K is the number of offspring.

After evaluating the fitness of all offspring K, the new mean is calculated as a weighted sum of the top performing candidates:

$$\mathbf{m}^{(g+1)} = \sum_{k=1}^{K} w_k \mathbf{x}_{k:K}^{(g+1)} \tag{6.2}$$

where w_k are the selection weights and $\mathbf{x}_{k:K}^{(g+1)}$ represents the k-th best individual in the sorted population.

CMA-ES dynamically adapts both the covariance matrix and the step-size over generations using evolution paths, helping to maintain a balance between exploration and exploitation. This adaptability allows CMA-ES to perform robustly on complex, noisy, and high-dimensional optimization problems. Its ability to optimize non-differentiable objective functions makes it a strong candidate for tasks such as the generation of test patterns in DNN hardware fault testing.

6.1.2. Fault Modeling

The fault models considered in this work are bit-flip faults, Gaussian perturbation faults, and stuck-at faults (see Section 2.3). These abstractions capture realistic hardware-induced errors in the weights and activations of DNNs accelerators.

- Bit-flip faults: A fraction p_w of network weights is randomly selected. For each selected weight $w \in \mathbb{R}$ with fixed-point encoding $b(w)$, a fraction p_b of its bits is flipped according to a chosen bit-position model, yielding

  $$\tilde{w} = \text{BitFlip}(w, p_b). \tag{6.3}$$

  Because bit positions have unequal numeric impact, three variants are considered: *BF-Random* (flip positions sampled uniformly over $b(w)$), *BF-MSB* (always flip the most significant bit), and *BF-LSB* (always flip the least significant bit).

- Gaussian Perturbation Faults: A fraction p_w of the weights is selected and perturbed by additive Gaussian noise. If σ is the standard deviation of the network weights, then for a selected weight w, the fault is injected as

  $$\tilde{w} = w + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \gamma \cdot \sigma) \tag{6.4}$$

  where γ is a scaling factor controlling the strength of the perturbation.

- Stuck-at faults: Two abstractions are considered. For per-weight stuck-at faults (SA0/1-W), individual weights are randomly selected and each selected w_{ij} is forced to a constant value of 0 or 1. For neuron-level stuck-at-0 faults (SA0-N), neurons are randomly selected and all outgoing weights w_{ij} from each selected neuron i are set to zero ($w_{ij} = 0 \; \forall j$), thereby suppressing its contribution to the next layer. Stuck-at-one neurons are not modeled explicitly, since in ReLU-based networks persistently high activations are already covered by strong positive perturbations in the other fault models.

Fault scenarios are generated by applying one of the fault models, producing an ensemble of faulty networks $\mathcal{F}$. Each $f_i \in \mathcal{F}$ is obtained by injecting a unique instance of faults of the same type into the fault-free model. This set $\mathcal{F}$ constitutes the reference pool on which candidate test patterns are evaluated during optimization.

6.2. Functional Testing and Compact Test Patterns Optimization

This section formulates the compact functional testing problem as a unified optimization task and describes the integration of test generation and compaction within a single objective.

Algorithm 3 Sequential Coverage-First then Compaction

Input: fault list, desired coverage, maximum rounds
Output: compact test suite
Initialise *uncovered faults* ← fault list, *suite* ← ∅
repeat
    Generate one test pattern for *each* fault in *uncovered faults*
    Add all generated patterns to *suite*
    Remove faults now covered from *uncovered faults*
until desired coverage reached **or** maximum rounds exhausted
Compact *suite* by removing redundant patterns
return *suite*

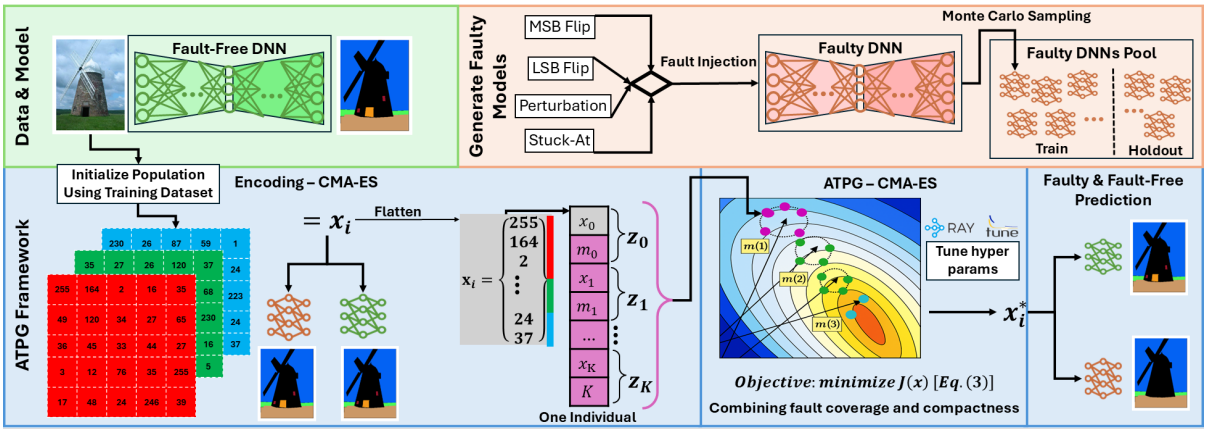

Figure 6.1.: CMA-ES-based ATPG workflow. From a fault-free DNN and dataset, we instantiate fault models and, via Monte Carlo sampling, build disjoint training and validation fault pools. In the ATPG stage, each individual encodes K candidate test patterns initialized from data, and CMA-ES optimizes a non-differentiable objective that maximizes fault coverage $\mathrm{cov}(X)$ while minimizing $|X|/K$ (compaction) under a test-set size limit K. Hyperparameters are tuned using Ray Tune [41]. Coverage is computed by comparing the predictions of faulty and fault-free networks.

The method aims to generate a compact set of functional test patterns that achieves high fault coverage in DNNs hardware accelerators. Although the underlying fault models are structural and originate from hardware, the testing approach is purely functional: each fault is evaluated exclusively through its effect on the observable input-output behavior of the network. A fault is considered detected only if it produces an incorrect or measurably deviated output at the accelerator interface.

For classification tasks, including semantic segmentation, a test pattern is counted as detecting a fault if the predicted labels differ between the fault-free and faulty networks. This black-box perspective is particularly practical for real deployment scenarios such as cloud or edge AI accelerators, where internal architectural details are inaccessible or proprietary. In such environments, behavioral deviations at the output interface constitute the only feasible means of fault detection.

Unlike previous approaches that first generate a large set of test patterns and subsequently apply compaction as a post-processing step (see Algorithm 3), the proposed framework integrates both objectives directly into a single optimization loop. The goal is to simultaneously

1. maximize fault coverage,
2. maximize output deviation, and
3. minimize test-set size.

This concurrent formulation eliminates the need for a second compaction pass and scales more effectively to large fault sets. The overall workflow is illustrated in Fig. 6.1.

Using the fault models introduced in Section 6.1.2, we construct a finite ensemble of faulty networks $\mathcal{F} = \{f_1, \ldots, f_n\}$ by injecting sampled faults into the fault-free network Net. Let $\mathrm{Net}_f(\mathbf{x})$ denote the network obtained by injecting fault $f \in \mathcal{F}$.

The unified objective is defined as

$$J(\mathbf{X}) = \big(1 - \mathrm{cov}(\mathbf{X})\big) \cdot \big(1 - \lambda_{\text{dev.}} \cdot \mathrm{diff}(\mathbf{X})\big) + \lambda_{\text{comp.}} \cdot |\mathbf{X}|. \tag{6.5}$$

where $\lambda_{\text{dev.}}, \lambda_{\text{comp.}} \in [0, 1]$ balance the relative importance of deviation detection and compactness. The objective is minimized.

Test-set size $|\mathbf{X}|$. Compactness is enforced by penalizing the number of active patterns. The count is normalized by the user-defined upper bound K

$$|\mathbf{X}| = \frac{\mathrm{active}(\mathbf{x})}{K}. \tag{6.6}$$

Fault coverage $\mathrm{cov}(\mathbf{X})$. Coverage measures the fraction of faults in $\mathcal{F}$ for which at least one test pattern produces a different class decision than the fault-free network

$$\mathrm{cov}(\mathbf{X}) = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \max_{\mathbf{x} \in \mathbf{X}} \mathbf{1}\left[\arg\max \mathrm{Net}_f(\mathbf{x}) \neq \arg\max \mathrm{Net}(\mathbf{x})\right]. \tag{6.7}$$

Output deviation $\mathrm{diff}(\mathbf{X})$. To encourage strong behavioral differences, the maximum ℓ_2 deviation between faulty and fault-free outputs is averaged across faults

$$\mathrm{diff}(\mathbf{X}) = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \max_{\mathbf{x} \in \mathbf{X}} \left\|\mathrm{Net}_f(\mathbf{x}) - \mathrm{Net}(\mathbf{x})\right\|_2. \tag{6.8}$$

Because coverage involves arg max operations and compaction involves discrete activation decisions, the resulting objective is non-differentiable. Consequently, gradient-based methods are not suitable for this joint optimization problem.

In this framework, compaction does not compress or modify the test patterns themselves. Instead, an activation-mask representation allows the optimizer to select which patterns remain active. Compaction therefore corresponds solely to minimizing the number of retained patterns while preserving functional coverage.

The following section reformulates the problem to enable optimization using a black-box evolutionary strategy based on CMA-ES.

6.3. Evolutionary Optimization Using Covariance Matrix Adaptation Evolution Strategy

To address the non-differentiability of the objective, CMA-ES, a population-based evolutionary optimizer well suited for high-dimensional, multimodal, and gradient-free search spaces, is employed.

6.3.1. Candidate Representation

Each individual encodes an entire test suite $\mathbf{X} \in \mathbb{R}^{K \times d}$, where K is the maximum number of patterns and d is the dimension of a single flattened input.

Algorithm 4 Compact Functional ATPG using CMA-ES

Require: Net, dataset D, injector INJECT, test-set size limit K; fault-pool sizes $N_{\text{tr}}, N_{\text{val}}$; search space $\mathcal{S}$ over $(\text{pop}, \sigma_0, \lambda_{\text{dev}}, \lambda_{\text{comp}}, \tau)$; trials N_{trial}; generations $G_{\text{tune}}, G_{\text{final}}$
Ensure: compact suite $X^\star$ with $|X^\star| \le K$

$\mathcal{F}_{\text{tr}} = \{\text{INJECT}(\textbf{Net}; \text{seed}_i)\}_{i=1}^{N_{\text{tr}}}$, $\mathcal{F}_{\text{val}} = \{\text{INJECT}(\textbf{Net}; \text{seed}_j)\}_{j=1}^{N_{\text{val}}}$
$(\theta^\star, J^\star) \leftarrow (\varnothing, +\infty)$
for $t = 1..N_{\text{trial}}$ **do** ▷ short CMA-ES runs
  sample $\theta \sim \mathcal{S}$; init mean m with K noisy samples from D
  for $g = 1..G_{\text{tune}}$ **do**
    sample $Z_j \sim \mathcal{N}(m, \sigma^2 C)$; decode X_j via τ (ensure $|X_j| \le K$)
    $J_j = (1 - \text{cov}(X_j; \mathcal{F}_{\text{tr}}))(1 - \lambda_{\text{dev}}\text{diff}(X_j; \mathcal{F}_{\text{tr}})) + \lambda_{\text{comp}} \frac{|X_j|}{K}$
    update (m, C, σ) using best μ (lowest J_j)
  end for
  $J \leftarrow \min_j J_j$; **if** $J < J^\star$ **then** $(\theta^\star, J^\star) \leftarrow (\theta, J)$
end for
Final run: $X^\star \leftarrow$ CMAES_DECODE(CMAES_RUN$(\theta^\star, \mathcal{F}_{\text{tr}}, G_{\text{final}})$)
return $X^\star$ (evaluate on $\mathcal{F}_{\text{val}}$ separately)

To maintain a fixed chromosome length while allowing a dynamic suite size, each pattern is represented with an activation mask

$$\mathbf{z}_k := [\mathbf{x}_k \mid m_k] \in [0, 1]^{(d+1)}. \tag{6.9}$$

The number of active patterns is defined as

$$|\mathbf{X}| := \sum_{k=1}^{K} \mathbf{1}_{\{m_k \ge \tau\}}. \tag{6.10}$$

This representation enables adaptive compaction during evolution without changing chromosome dimensionality.

Population initialization leverages training samples as priors. Because networks are optimized for training distributions, perturbations around those inputs are more likely to expose behavioral discrepancies between fault-free and faulty models.

6.3.2. Optimization Procedure

CMA-ES iteratively samples candidate solutions from a multivariate Gaussian distribution

$$\mathbf{z} \sim \mathcal{N}(\mathbf{m}, \Sigma),$$

where $\mathbf{m}$ and Σ are updated based on fitness ranking.

The initial mean is seeded from training samples with Gaussian perturbations. Mask variables are initialized to favor sparsity.

A two-stage hyperparameter tuning strategy is employed. First, candidate configurations are evaluated with limited generations and early stopping. The best configuration is then refined in a second stage with extended generations.

Each generation has complexity

$$O(|\text{pop}|d^2 + |\text{pop}|Kd),$$

where |pop| is the population size.

Table 6.1.: Benchmark DNNs and datasets across convolutional, residual, transformer, and encoder–decoder architectures; parameter counts in millions (M).

Model / Dataset	Arch. type	Params [M]
CNN-8L / CIFAR-10 [20]	convolutional CNN	≈1.29
ResNet-19 [31] / Tiny-ImageNet [29]	residual CNN	≈11.28
ResNet-20 / CIFAR-10 [99]	residual CNN	≈0.27
ViT-Tiny [65] / SVHN [21]	transformer	≈5.53
ResNet-50 + PPM-Deeplab [38] / ADE20K[39]	encoder–decoder	≈51.70

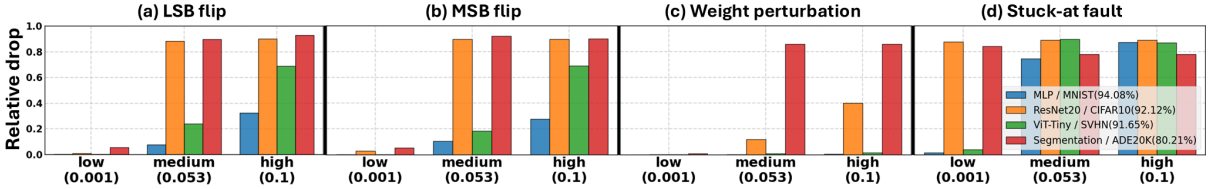

Figure 6.2.: Relative accuracy drop across four DNN architectures under four fault models. Each architecture is evaluated at three severities: low (0.001), medium (0.053), and high (0.1), denoting the affected-weight ratio.

Although CMA-ES does not guarantee global optimality, it is highly effective for non-differentiable objectives such as the integrated compact test generation problem considered here.

6.4. Experimental Evaluation and Analysis

6.4.1. Experimental Setup

The proposed evolutionary optimization framework is compared against a gradient-based baseline, implemented as described in Algorithm 3. All experiments use identical fault-injection settings (Section 6.1.2) and operate under equal resource constraints to ensure a fair comparison.

Architectures and Datasets. The evaluation covers five pretrained DNN architectures from four architectural families: convolutional CNNs, residual CNNs, transformers, and encoder–decoder segmentation networks. The experiments span image classification (CIFAR-10, Tiny-ImageNet, SVHN) and semantic segmentation (ADE20K). The selected models vary significantly in depth and parameter count, enabling assessment of scalability across network complexity (see Table 6.1).

Hardware and Software. All experiments were executed on the HoreKa Green GPU partition at NHR@KIT using a single NVIDIA A100 GPU (40 GB) and an Intel Xeon Platinum 8368 CPU. The software stack includes PyTorch 2.2.2 [49], EvoTorch 0.5.1 [85], and Ray Tune with Optuna 4.3.0 [43] for hyperparameter optimization.

6.4.2. Results

Fig. 6.2 illustrates the relative accuracy degradation of four representative architectures under LSB/MSB bit flips, weight perturbations, and stuck-at faults across three affected-weight ratios (0.001, 0.053, 0.1). ResNet-18 (classification) and ResNet-50 (segmentation) exhibit the most pronounced degradation, whereas the MLP remains comparatively robust. Weight perturbations cause gradual performance degradation as severity increases, while stuck-at faults induce the largest accuracy drops at each severity level.

Table 6.2 reports average fault coverage and test-suite size over the training fault pool F_{tr}. For CNN-8L, the proposed CMA-ES-based method improves coverage from 92.3% to 96.7% while reducing the average

Table 6.2.: Summary of coverage and test-suite size per architecture. Fault coverage (Cov%) and number of test patterns (#TP) are averaged over all fault models and severities on the training fault pool (F_{tr}). Values in parentheses in the $\#\text{TP}_{\text{prop}}$ column denote the compaction factor ($\#\text{TP}_{\text{base}}/\#\text{TP}_{\text{prop}}$). All reported #TP values respect the test-set size limit (K).

	Cov%		#TP	
Arch.	Prop.	Grad. [81]	Prop.$\leq K$ (Comp.)	Grad. [81]
CNN-8L	96.7 ± 2.3	92.3 ± 5.3	3≤ 30 (6.3×)	19
ResNet19	100 ± 0.0	100 ± 0.0	2≤ 30 (3.5×)	7
ResNet20	100 ± 0.0	100 ± 0.0	6≤ 30 (16.3×)	98
ViT-Tiny	100 ± 0.0	100 ± 0.0	4≤ 10 (3×)	12
ADE20K	97.5 ± 0.0	96.3 ± 6.5	3≤ 10 (2.6×)	8
Average	**98.7** ± **2.12**	97.5 ± 5.0	**3.2** (9×)	28.8

Table 6.3.: Generalization of the CMA-ES-based method compared to the gradient-based method (Grad.) [81]. Coverage is measured on a validation fault pool F_{val} disjoint from F_{tr}. Execution time per fault is the average time to evaluate one faulty DNN (in seconds) in F_{val}, and suite size is the total stored test-set size in megabytes (MB).

DNN arch.	Fault model	Cov%		Exec./fault [s]		Suite size [MB]	
		Prop.	Grad.	Prop.	Grad.	Prop.	Grad.
ResNet20	BF-LSB	**96.7**	70.0	**0.02**	0.17	**0.10**	0.99
	BF-MSB	**100**	70.0	**0.01**	0.17	**0.08**	0.99
	WP	**100**	43.3	**0.02**	0.24	**0.09**	1.44
	SA0/1-W	**100**	**100**	**0.01**	0.20	**0.03**	1.16

BF-LSB/BF-MSB: bit flip at LSB/MSB; WP: weight perturbation (Gaussian); SA0-N: neuron stuck-at-0.

suite size from 19 to 3 patterns (6.3× compaction). For ResNet19, ResNet20, ViT-Tiny, and ADE20K, coverage remains comparable to the gradient-based baseline (100%, 100%, 100%, and 97.5%), yet pattern counts decrease significantly (7→2, 98→6, 12→4, and 8→3, respectively), all within predefined test-set limits.

Table 6.3 evaluates generalization on a held-out validation fault pool F_{val}, consisting of 30 faulty ResNet20 instances per fault model not used during optimization. For bit-flip (LSB/MSB) and weight-perturbation faults, the proposed method achieves 96.7–100% coverage on unseen faults, substantially outperforming the gradient-based baseline (43.3–70.0%). Simultaneously, execution time per faulty model decreases from 0.17–0.24 s to 0.01–0.02 s, and suite size shrinks from approximately 1 MB to below 0.1 MB. For stuck-at faults, both methods achieve full coverage; however, the evolutionary suite remains smaller and faster to execute.

6.4.3. Discussion

Fig. 6.2 indicates that deeper residual networks exhibit significantly larger performance gaps between low and high severities compared to the MLP. This suggests that, at a fixed affected-weight ratio, cumulative fault effects scale with model depth and parameter count. The similar behavior observed for LSB and MSB bit flips can be attributed to the prevalence of small-magnitude weights and normalized activations (e.g., batch normalization and ReLU), which reduce the destructive impact of many nominal MSB flips. Consequently, compact test suites that effectively expose generic bit-flip faults are typically sufficient to detect both LSB and MSB errors, although inferring the precise flipped bit from functional outputs remains challenging.

Table 6.2 demonstrates the effectiveness of directly optimizing compact test suites under explicit size constraints. The CMA-ES-based method matches or exceeds gradient-based coverage while using on

average nine times fewer patterns. By optimizing a non-differentiable objective that jointly accounts for coverage and compactness, redundant patterns are eliminated during search rather than through post-hoc compaction. As a result, near-perfect coverage is achieved with only a small number of patterns per architecture.

The validation results in Table 6.3 further show that optimization on a richer training fault pool F_{tr} yields suites that generalize effectively to unseen faults. In contrast, gradient-based suites exhibit substantial performance degradation on random faults, particularly under weight perturbations. This highlights the advantage of explicitly balancing coverage and compactness during optimization.

From a computational perspective, the evolutionary approach incurs higher offline generation time—on the order of ten hours per architecture—due to population-based sampling and repeated fault evaluation. However, this cost is incurred only once during test generation. During deployment, in-field testing requires evaluating only a small number of compact patterns (typically 1–10 per model), resulting in negligible runtime and storage overhead. Thus, the method trades increased offline optimization effort for significantly reduced in-field testing cost.

6.5. Chapter Summary

This chapter presented an evolutionary optimization framework for compact functional testing of DNN accelerators. In contrast to gradient-based approaches introduced in earlier chapters, the proposed method formulates test pattern generation and compaction as a unified, non-differentiable optimization problem. By jointly maximizing fault coverage and minimizing test set size under explicit constraints, the framework directly produces compact and effective test suites without requiring post-hoc compaction.

Across diverse architectures and fault models, the CMA-ES-based approach achieved an average fault coverage of 98.7% while reducing the number of required test patterns by approximately 9× compared to a gradient-based baseline. Moreover, the generated test suites demonstrated strong generalization to unseen faults and significantly reduced in-field execution time and storage requirements.

These results indicate that evolutionary strategies provide a scalable and flexible foundation for functional hardware testing of DNN accelerators, particularly when compactness and non-differentiability must be addressed simultaneously.

7. Storage-Efficient Functional Test Generation

7.1. Chapter Overview

The previous chapters focused on generating compact and high-coverage functional test patterns for DNN accelerators. While reducing the number of test patterns lowers execution overhead, practical in-field deployment introduces an additional constraint: test patterns must be stored, transferred, and executed under strict memory and latency budgets. In many edge and embedded Artificial Intelligence (AI) platforms, memory capacity is limited, and the storage of even moderately sized test patterns can become prohibitive.

This chapter addresses the problem of storage-efficient functional testing for DNNs accelerators. Rather than solely minimizing the number of test patterns, it is investigated how test patterns can be represented and generated in ways that significantly reduce storage overhead while preserving high fault coverage. The central question is therefore not only how to generate effective test inputs, but also how to deploy them efficiently in resource-constrained environments.

Two strategies are presented. First, NBIST introduces a lightweight generator network co-deployed with the Network Under Test (NUT). Instead of storing raw test patterns, NBIST stores only the generator parameters and a seed value, enabling on-demand generation of structured test inputs directly on the inference engine. This approach transfers the principles of classical BIST from structural digital testing to the domain of neural inference, enabling autonomous validation in the field without requiring hardware modifications.

Second, a basis-driven test pattern representation framework expresses test inputs as linear combinations of basis vectors. Rather than storing complete test patterns, only optimized combination coefficients are retained within a predefined storage budget. This formulation integrates storage constraints directly into the test generation process and provides an analytical alternative to neural generative compression.

Together, these approaches explore nonlinear (generator-based) and linear (basis-based) representations for storage-efficient test deployment, offering distinct design trade-offs in terms of flexibility, scalability, and implementation complexity.

The remainder of this chapter is organized as follows. Section 7.2 presents the conceptual design of the NBIST framework. Section 7.3 formalizes the probabilistic fault modeling and generator training procedure underlying NBIST, followed by its experimental evaluation in Section 7.4. Section 7.5 introduces the basis-driven storage-aware test generation framework. Section 7.5.2 presents the basis-driven storage-constrained test generation method, and Section 7.6 presents the corresponding experimental evaluation. Finally, the chapter concludes with a unified discussion and summary comparing both approaches.

7.2. Neural Built-In Self-Test: Concept and System Overview

Inspired by classical BIST techniques in digital circuit testing, a functional self-test mechanism tailored to DNNs accelerators is introduced. Traditional BIST approaches employ compact on-chip pattern generators to produce deterministic, high-coverage structural test patterns without requiring large external storage. Techniques such as store-and-generate [5], bit-fixing [7], and bit-flipping [8] transform pseudorandom sequences into effective test stimuli, enabling autonomous and repeatable testing directly within the

hardware. These principles have proven highly successful in reducing storage overhead while maintaining high fault coverage in conventional digital systems.

As DNNs increasingly serve as the computational backbone of modern AI applications, including medical diagnosis, fraud detection, and autonomous driving [37], [70], they are deployed on specialized inference platforms such as GPUs, TPUs, and Application-Specific Integrated Circuits (ASICs). Although these platforms undergo structural manufacturing tests, in-field operation exposes them to additional reliability threats, including radiation-induced bit flips, aging-related degradation, and escaped manufacturing defects [3]. Such faults may silently corrupt inference results and lead to incorrect predictions in safety-critical environments.

Functional testing provides a complementary validation mechanism by evaluating the observable input-output behavior of DNNs under fault conditions [46], [78]. Unlike structural testing, which verifies circuit-level correctness, functional testing activates the learned inference paths of the model and determines whether hardware faults manifest as prediction errors. However, conventional functional testing approaches rely on the storage of large sets of precomputed test inputs. Even when compacted, these test patterns incur storage costs proportional to the number and dimensionality of the test patterns, making them unsuitable for memory-constrained edge deployments and autonomous in-field validation [80], [93].

To address this limitation, the NBIST approach is proposed, in which a lightweight generator network is co-deployed with the NUT. Instead of storing raw test patterns, NBIST stores only the generator parameters, a seed value from a pseudorandom number generator, and the expected output labels. During self-test operation, structured test inputs are generated on demand by mapping seeded random vectors through the trained generator network. This approach transfers the concept of built-in pattern generation from structural BIST to functional DNN inference testing.

By integrating the generator within the same inference engine as the NUT, NBIST enables autonomous, repeatable, and storage-efficient in-field testing without hardware modification. Unlike purely random testing, the generator is trained to produce fault-sensitive inputs, thereby achieving high fault coverage with a small number of generated patterns. This results in a storage requirement that is independent of the number of generated test patterns, i.e., it does not scale with the effective test set size.

Extensive experimental evaluations across multiple DNNs architectures, task types, and fault models demonstrate that NBIST achieves high fault coverage—up to 100% in several configurations—while significantly reducing memory overhead compared to conventional pre-stored functional test sets.

The remainder of this section is organized as follows. Section 7.3 presents the NBIST methodology and the training procedure. Section 7.5.1 describes the adopted fault model. Section 7.4 provides the experimental evaluation.

7.3. Neural Built-In Self-Test for Probabilistic Fault Models

To address the storage and in-field deployment constraints discussed earlier, NBIST is proposed as a functional testing framework that integrates test generation directly into the inference engine. NBIST operates entirely in software and executes on the same hardware as the NUT, requiring no additional circuitry or host-side control. This aligns with the built-in testing paradigm by enabling autonomous, self-contained functional validation.

The proposed framework consists of three components:

1. A seeded random number generator (RNG) that produces a pseudo-random sequence of vectors $\mathcal{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$.
2. A lightweight Test Pattern Generation Network (TPGNet), a small NN trained to map latent vectors $\mathbf{z}_n$ to structured test inputs $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$.
3. A set of expected labels $\{y_1, \ldots, y_N\}$ corresponding to the fault-free outputs of the NUT.

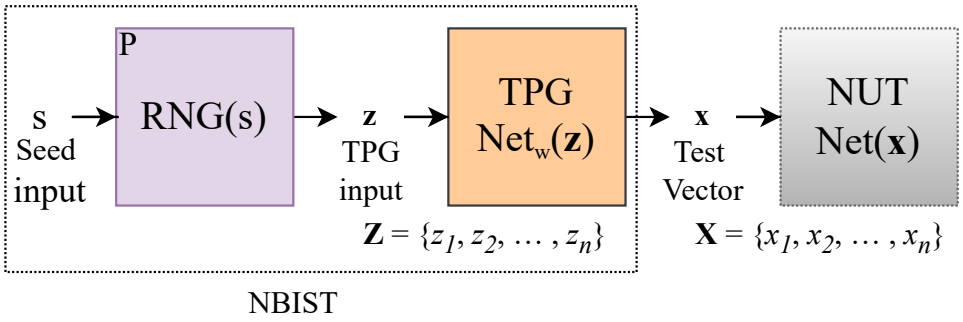

Figure 7.1.: Illustration of the NBIST framework. A seeded RNG produces vectors that are mapped by TPGNet into structured test patterns applied to the NUT.

This design enables deterministic on-demand reconstruction of the entire test suite $\mathcal{X}$ using only the seed, the TPGNet parameters, and the stored labels. Unlike conventional approaches that store raw test patterns, NBIST stores only the compact generator weights and a seed value, significantly reducing memory requirements.

The TPGNet can be co-deployed with the NUT on the same DNN accelerator, as illustrated in Fig. 7.1. During self-test operation, seeded vectors are mapped through TPGNet to generate structured test inputs, which are then applied to the NUT.

7.3.1. Training the Test Pattern Generation Network

For a given NUT, the TPGNet learns a mapping from latent vectors to test inputs,

$$\mathbf{x} = \mathrm{TPGNet}(\mathbf{w}, \mathbf{z}), \tag{7.1}$$

where $\mathbf{w}$ denotes the generator parameters and $\mathbf{z} = \mathrm{RNG}(s)$ is a vector derived from the seed s.

The objective is to train $\mathbf{w}$ such that the generated test patterns reveal faults in the NUT with high probability under a probabilistic fault model.

To quantify fault sensitivity, the discrepancy metric introduced in [80] is adopted,

$$d(\mathbf{x}, f) = \left\| \mathrm{Softmax}(\mathrm{Net}(\mathbf{x})) - \mathrm{Softmax}(\mathrm{Net}_f(\mathbf{x})) \right\|_2^2, \tag{7.2}$$

where Net denotes the fault-free network and Net_f denotes the network under fault instance f.

Faults are modeled as random transformations sampled from a distribution $f \sim p(f)$ (see Section 2.3). The goal is to optimize the generator parameters $\mathbf{w}$ to maximize expected fault sensitivity under this probabilistic model.

The training objective is defined as

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{f \sim p(f)} \left[\max_{\mathbf{z}_n \in \mathcal{Z}} d\big(\mathrm{TPGNet}(\mathbf{w}, \mathbf{z}_n), f \big) \right]. \tag{7.3}$$

The inner max operator encourages specialization: each generated pattern $\mathbf{x}_n$ focuses on faults to which it is most sensitive, while the set $\mathcal{Z}$ jointly provides coverage across the distribution $p(f)$.

In practice, the expectation is approximated via Monte Carlo sampling,

$$\mathcal{L}(\mathbf{w}) \approx \frac{1}{M} \sum_{m=1}^{M} \max_{\mathbf{z}_n \in \mathcal{Z}} d\big(\mathrm{TPGNet}(\mathbf{w}, \mathbf{z}_n), f^{(m)} \big), \quad f^{(m)} \sim p(f). \tag{7.4}$$

The generator parameters are optimized using gradient ascent.

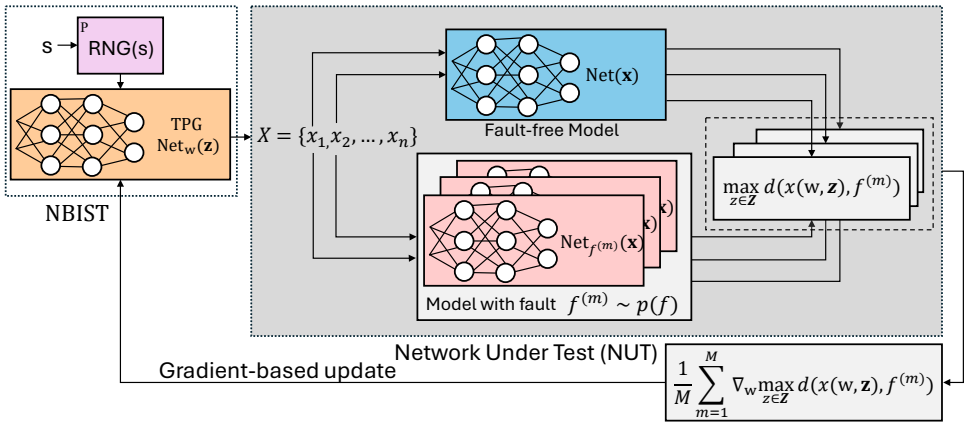

Figure 7.2.: Conceptual diagram of the TPGNet training procedure. A seeded RNG maps the integer seed s to a vector $\mathbf{z}$, which serves as input to TPGNet. Fault samples are drawn from a probabilistic fault model, and TPGNet is trained to produce test patterns that highlight the differences between the fault-free and faulty networks.

Importantly, although training uses multiple sampled faults and vectors, deployment requires storing only:

- the trained generator parameters $\mathbf{w}$
- a single seed s
- the expected labels y_n

This enables deterministic regeneration of the full test suite with negligible memory overhead.

Fig. 7.2 outlines the TPGNet training procedure, where the generator is trained through gradient ascent to generate a diverse test set that maximizes the expected fault sensitivity under the composite fault model. The overall pipeline is illustrated in Algorithm 5.

7.3.2. Test Pattern Generation Network Architecture Selection

To identify a suitable generator architecture, hyperparameter search methods such as random search [22] are employed across networks of varying depth and width.

The objective is to determine the smallest architecture that achieves strong fault coverage under a tight storage budget.

The TPGNet input dimension must match the vector size (e.g., 4), while the output dimension must match the flattened input dimension of the NUT (e.g., $3 \times 32 \times 32$ for CIFAR-10).

The training complexity scales as $O(N \times M)$ with respect to the number of vectors and sampled faults. At deployment time, inference requires only a forward pass per generated pattern.

In practice, the additional storage overhead is limited to the generator parameters (typically tens of kilobytes) and the seed value, making runtime and memory overhead negligible relative to the target DNN.

7.4. Experimental Evaluation

7.4.1. Composite Fault Model

The composite probabilistic fault model introduced in Section 2.3 is adopted. This model jointly applies multiple fault mechanisms to emulate realistic hardware degradation during in-field operation. Specifically, the following mechanisms are applied sequentially:

Algorithm 5 Training the TPGNet using gradient-ascent.

Input: Network under test Net; fault distribution $p(f)$; number of test patterns N; learning rate η; training steps T; seed s; σ hyper-parameter for initializing weights.
Select TPGNet architecture via random search and initialize weights $\mathbf{w} \sim \mathcal{N}(0, \sigma^2)$
Generate $\mathcal{Z} = \{\mathbf{z}_n \leftarrow \mathrm{RNG}(s)\}_{n=1}^{N}$
for $t = 1$ to T **do**
  for each vector $\mathbf{z}_n$ **do**
    Generate test input: $\mathbf{x}_n = \mathrm{TPGNet}(\mathbf{w}, \mathbf{z}_n)$
    Sample M faults: $f^{(m)} \sim p(f),\ m = 1, \ldots, M$
    Estimate $\mathcal{L}(\mathbf{w})$ using $f^{(m)}$ as in Eq. (7.4)
    Estimate gradient $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ using $f^{(m)}$
    Update TPGNet weights: $\mathbf{w} \leftarrow \mathbf{w} + \eta\nabla_{\mathbf{w}}\mathcal{L}$
  end for
end for
Return: trained TPGNet weights $\mathbf{w}$

- weight perturbations (Gaussian noise),
- bit-level flips, and
- stuck-at faults.

The composite formulation captures interactions between transient and permanent fault behaviors and provides a comprehensive assessment of fault sensitivity in practical AI hardware systems.

7.4.2. Datasets and Experimental Setup

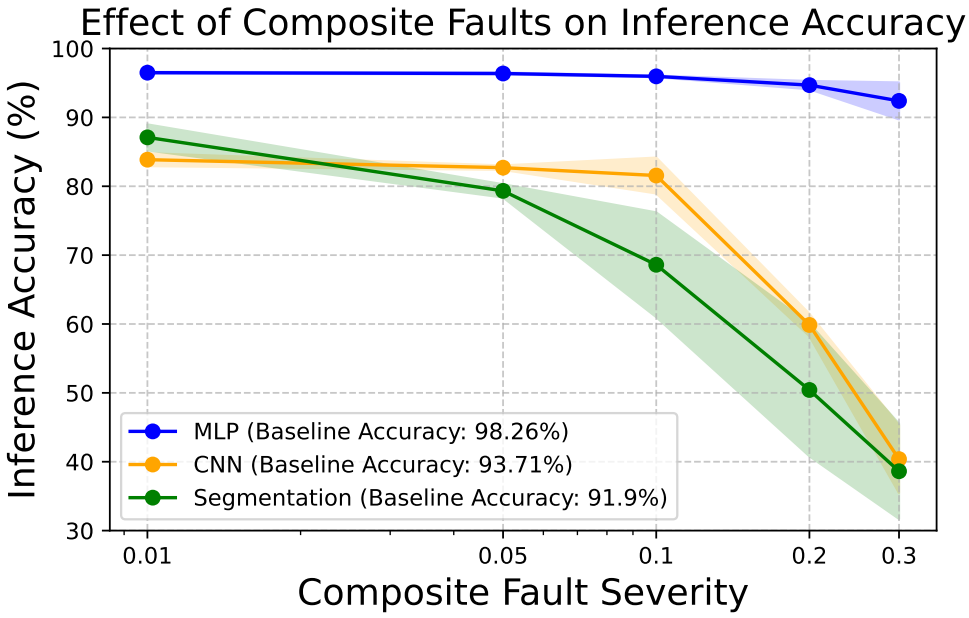

Figure 7.3.: Effect of composite faults on inference accuracy across different models. The composite fault model combines stuck-at-0, bit-flip, and Gaussian perturbation faults. The x-axis represents increasing fault severity, where the standard deviation (σ) of the Gaussian component controls the perturbation strength.

Fault coverage is evaluated across multiple DNN architectures and task types.

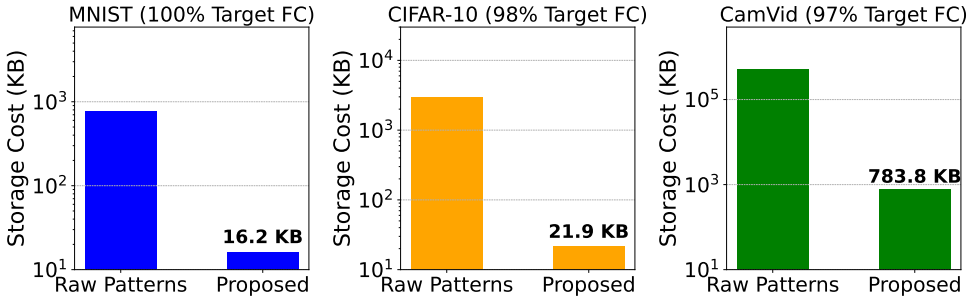

Figure 7.4.: Storage required to achieve target fault coverage under composite faults. "Raw" corresponds to direct storage of 1000 test inputs, while "Proposed" indicates storage of generator parameters only.

Table 7.1.: Comparison of fault coverage (%) and storage cost across different models, datasets, and test pattern generation strategies. The storage cost is separated into (i) raw storage of test patterns and (ii) the proposed NBIST approach, which includes the fixed parameters overhead and label cost. In the fault modeling context, p denotes the perturbation percentage, σ is the standard deviation of the added Gaussian noise, and ρ represents the percentage of flipped bits and stuck-at-zero faults.

Experimental setup			**Fault Coverage (%)**			**Storage Cost (KBytes)**		
Model and Dataset	**Composite Fault model**	**Number of Test patterns (N)**	**Random**	**Adv. [45]**	**Proposed**	**Raw Storage for N test patterns**	**Proposed NBIST**	
							Parameters Overhead	**Labels cost**
MLP (MNIST)	p=5% σ=0.01 ρ=0.05	10	11.5	40.0	97.0	7.7	18.2	0.10
		20	18.0	70.0	97.5	15.3	18.3	0.20
		50	26.0	90.0	98.5	38.3	18.0	0.49
		100	31.0	99.0	100.0	76.6	18.2	0.98
		200	39.5	100.0	100.0	153.1	18.3	1.95
ConvNet-7 (CIFAR-10)	p=5% σ=0.01 ρ=0.01	10	30.0	42.5	95.0	30.0	22.0	0.10
		20	32.0	46.5	96.0	60.0	21.8	0.20
		50	40.0	52.5	98.0	150.0	22.0	0.49
		100	42.0	54.5	98.0	300.0	22.0	0.98
		200	52.5	60.5	100.0	600.0	22.0	1.95
		1000	98.0	99.0	100.0	3000.0	21.9	9.77
VGG-19 (CIFAR-10)	p=15% σ=0.1 ρ=0.1	50	16.9	33.6	97.0	150.0	213.0	0.49
		100	35.8	58.2	100.0	300.0	250.0	0.98
CNN-based Segmentation (CamVid)	p=10% σ=0.01 ρ=0.01	10	0.0	15.5	95.0	480.0	210.1	160.0
		20	6.0	30.7	100.0	960.0	240.4	320.0
		50	22.2	52.1	100.0	2400.0	608.3	800.0
		100	40.0	60.0	100.0	4800.0	717.0	1600.0
		1000	97.0	97.8	100.0	506,250.0	783.8	16,000.0

For image classification tasks, the following models are considered:

- an MLP,
- a custom 7-layer convolutional network,
- Visual Geometry Group (VGG)-19 [30],

trained on MNIST [24] and CIFAR-10 [25].

For semantic segmentation, a custom 10-layer convolutional model trained on CamVid [18] is employed, including convolutional, pooling, and upsampling layers for pixel-wise classification.

To identify a suitable TPGNet architecture, random search [22] is applied to fully connected networks with three hidden layers (4 to 32 units each). Training is performed using the Adam optimizer [27] for 100 iterations. The learning rate is selected through grid search on a logarithmic scale from 10^{-7} to 1.

At each optimization step, the objective and gradients are estimated using Monte Carlo sampling with $M = 10$ fault samples drawn from $p(f)$.

During evaluation, a seed s generates vectors $\mathbf{z}_n$, which are mapped to test inputs:

$$\mathbf{x}_n = \mathrm{TPGNet}(\mathbf{w}, \mathbf{z}_n).$$

Each generated input $\mathbf{x}_n$ is passed through the NUT to obtain

$$\hat{y}_n = \arg\max_c \mathrm{Net}(\mathbf{x}_n)_c.$$

The predicted label $\hat{y}_n$ is compared against the stored reference label y_n. A fault is considered detected if any $\hat{y}_n \neq y_n$.

Fault coverage is estimated using $M = 1000$ sampled faults. Each experiment is repeated five times using independent random seeds.

The storage requirements differ between testing strategies:

Raw Test Pattern Storage

$$\mathrm{Storage}_{\mathrm{Raw}} = N \times (D + L),$$

where D is the input size (bytes) and L is the label size.

NBIST Storage

$$\mathrm{Storage}_{\mathrm{NBIST}} = \underbrace{\text{Generator Size}}_{\text{Fixed Overhead}} + \underbrace{N \times L}_{\text{Label Storage}} + \underbrace{\text{Seed}}_{\text{One Integer}}.$$

Label storage depends on the task type.

For MNIST and CIFAR-10, each label is a single class index (1 byte), resulting in negligible overhead (1 kB for 1000 patterns).

For segmentation datasets such as CamVid, labels are pixel-wise masks. Even when downsampled (e.g., 128×128), each mask requires 16 kB per sample (1 byte per pixel), significantly increasing the storage cost.

This illustrates that storage efficiency becomes even more critical for dense prediction tasks.

All experiments are implemented in PyTorch [49] with 32-bit precision. MLP experiments are executed on CPU, while CNN and segmentation models are trained and evaluated on an NVIDIA A100 GPU.

7.4.3. Results and Discussion

In this section, the results obtained from the conducted experiments are discussed. Fig. 7.3 illustrates inference degradation under increasing composite fault severity. The MLP (baseline accuracy 98.26%) exhibits the highest robustness, maintaining stable accuracy as σ increases. In contrast, the CNN (93.71%) and the segmentation model (91.9%) show more pronounced degradation at moderate fault levels (0.1–0.2), reflecting increased sensitivity due to greater depth and parameter complexity.

These results confirm that deeper architectures are more susceptible to weight perturbations and bit-level errors under identical fault conditions.

To assess storage efficiency, the memory required to achieve the target fault coverage under composite fault injection ($N = 1000$) is measured, as shown in Fig. 7.4.

Across MNIST (100% coverage), CIFAR-10 (98%), and CamVid (97%), NBIST consistently achieves the target coverage with substantially lower storage requirements than pre-stored or adversarially generated test sets.

The memory savings become increasingly significant for larger models and higher-dimensional inputs.

Table 7.1 summarizes coverage and storage across models and test strategies.

NBIST consistently outperforms random and adversarial baselines in coverage while reducing memory usage by one to three orders of magnitude.

For example, in the MNIST MLP experiment with $N = 10$ patterns:

- Random testing achieves 11.5% coverage.
- NBIST achieves 97.0% coverage using only 18.2 kB of generator parameters and 0.1 kB of label storage.

For CIFAR-10 with $N = 1000$, NBIST achieves 100% coverage using less than 800 kB of total storage, compared to more than 506 MB required for storing raw test inputs.

This demonstrates that learned, on-demand generation provides a scalable and storage-efficient alternative to pre-stored test sets.

In-field execution requires only a forward pass through TPGNet per test pattern. Although generation incurs a slightly higher per-pattern cost than random sampling, NBIST achieves the target coverage with far fewer test vectors.

Consequently, the total execution time—dominated by NUT inference—is significantly reduced compared to random or adversarial approaches, especially for larger N.

NBIST scalability depends primarily on input dimensionality rather than model depth. The generator is trained once and reused across all patterns, so its parameter size remains constant as N increases.

Training complexity scales as $O(N \times M)$, while inference cost per pattern remains constant.

In general, the results demonstrate that NBIST effectively transfers the autonomy of classical BIST into functional DNN testing. It achieves high fault coverage without additional hardware and with minimal storage overhead. Although label storage is required, this cost is negligible for classification tasks and remains manageable even for segmentation scenarios.

7.5. Basis-Driven Storage-Aware Test Generation

In this section, a storage-aware functional test generation approach based on compact linear representations of test patterns is presented. While the previous section introduced a neural generator network for on-demand test creation, the method proposed here follows a different philosophy: instead of learning a nonlinear generator, test patterns are constructed from a shared set of reusable basis components.

In practical in-field testing scenarios, storage is often the primary constraint [47], [80]. High-dimensional inputs, such as images, require substantial memory when stored directly, and even a modest number of test patterns can exceed the available memory budget on edge or embedded AI platforms. This challenge becomes more pronounced as DNNs continue to grow in size and are increasingly deployed on specialized accelerators such as systolic arrays and memristive crossbars [2], [40]. Although these architectures significantly improve inference efficiency, their reliability must be ensured through functional testing, both during manufacturing and in-field operation [50]. Achieving high fault coverage typically requires a large number of test patterns, which directly translates into high storage overhead.

Traditional approaches first generate full-resolution test inputs and then apply compression, for example, through post-generation compaction techniques [63], [80]. However, post-processing compression does not guarantee that the final test set satisfies strict storage limits, since storage constraints are not explicitly considered during test generation.

The key idea of the proposed method is to represent each test pattern in a compact form using a predefined set of basis vectors. Rather than storing complete input tensors, only a small set of coefficients describing how the basis elements are combined must be retained. The full test patterns can then be reconstructed on demand by recombining these shared basis elements during test execution. This linear representation leverages the fact that many fault-revealing inputs lie in structured subspaces of the input domain, allowing efficient parameterization without storing every pixel explicitly.

This approach integrates storage considerations directly into the test generation phase. By optimizing test patterns in their compressed representation, the method ensures that the resulting test suite meets predefined memory constraints while still achieving high fault coverage. In contrast to hardware-dependent structural DFT techniques, such as scan-based or structural BIST, the proposed framework operates purely at the functional level and is independent of the underlying accelerator architecture. It can therefore be applied across different deployment platforms, ranging from GPUs and TPUs to custom AI accelerators.

The remainder of this section is organized as follows. Section 7.5.1 describes the fault model used in this work. Section 7.5.2 details the basis-driven representation and explains how storage constraints are incorporated into the optimization procedure. Section 7.6 presents the experimental results.

7.5.1. Fault Modeling

A classical fault model for testing digital circuits on the logic-level are SAF [16]. For NNs, a definition of SAF on the neuron level can be expressed as

$$\text{Neuron}_j^l(\mathbf{x}) = v \quad \forall \mathbf{x} \in \mathbb{X}.$$

Here, a neuron $\text{Neuron}_j^l(\mathbf{x})$ (index j of some layer l) is defined as stuck-at a fixed, deterministic value $v \in \mathbb{R}$. The specific choices for v depend on the activation function $\phi(\cdot)$ and are chosen as signature values from its range. Specifically, the maximum or the minimum values, e.g, 0 and 1 for the sigmoid activation function.

The rational behind these choices is that any neuron internal faults in the weights, the calculation of the sum (MAC operation) or the activation function, if not masked, should affect the neuron output and its firing. Therefore, similar to the pin faults in digital testing, we consider neuron faults as an abstraction of faults in the internal components of the neuron (see Fig 7.5).

When performing fault simulation at the weights by injecting multiple weight faults at the bit level, we can observe how these faults (inside) the neuron show up as a neuron SAF, proving the relevance of the SAF on the neuron level. Also, modeling weight faults can be computationally costly as the total weight parameters are orders of magnitude larger than the number of neurons in a typical DNN. In contrast, neuron faults and their modeling are computationally tractable. We also experimentally validate (see Sec. 7.6) that weight stuck-at faults can either be masked (i.e., redundant faults) or result in stuck-at faults in neuron outputs. Analogously to neurons, the concept of SAF can also be extended to filters in CNNs. In case of a faulty filter, the SAF affects the entire filter map (output channel), resulting in a constant channel value of v.

For simplicity, we refer to a (stuck-at) fault simply as f. A fault free NN is denoted by $\text{Net}(\cdot)$, while a faulty network suffering from a (stuck-at) faults f is denoted by $\text{Net}_f(\cdot)$. Additionally, we denote the set $\mathcal{F} := \left\{f^{(1)}, \cdots, f^{(n)}, \cdots, f^{(N)}\right\}$, as a fault list, containing the faults of interest, for which test patterns should be generated. Also, we will use the shorthand notation n_j for the neurons.

Note that the proposed approach is generic and independent of the particular fault model and can be applied to both single and multiple faults at weights or neuron level. However, due to the nature of stuck-at faults, a high number of required test patterns can be expected (potentially one per neuron), which especially drives the need for storage efficient representations.

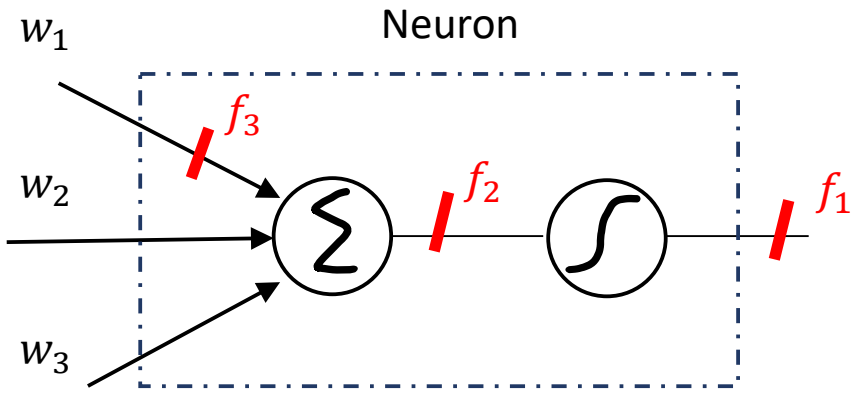

Figure 7.5.: The fault analysis on how the fault (inside) the neuron (f_3 at the weights, f_2 the outputs of the MAC, and f_1 the outputs of activation function) represent a neuron fault.

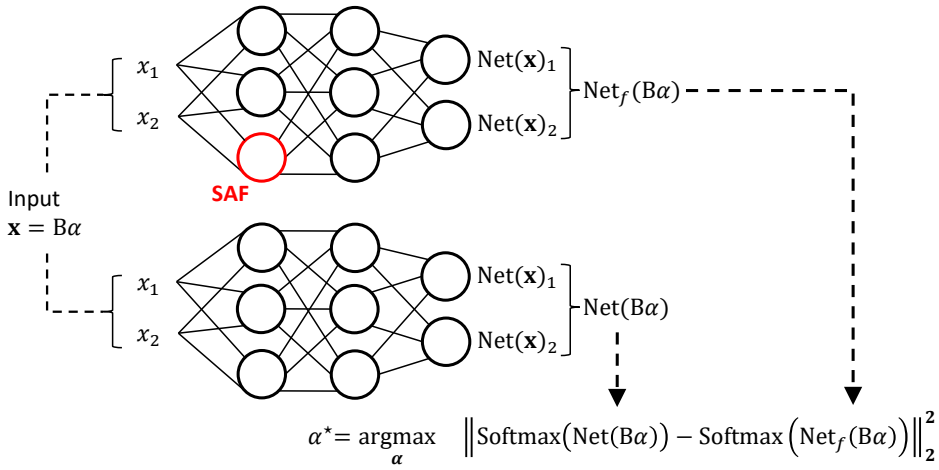

Figure 7.6.: An illustration of the objective where the test pattern is constructed as a linear combination of a joint D-dimensional basis $\mathbf{B}$ and its coefficient vectors $\boldsymbol{\alpha}$.

7.5.2. Test pattern generation

For testing a NN, we assume that a fault cannot be directly assessed. We may only control and apply given inputs $\mathbf{x}$ and observe the respective resulting outputs $\mathrm{Net}(\mathbf{x})$ of the NN. Additionally, a fault may only be revealed if the prediction, i.e. the index with the maximum output value, is different for the fault-free $\mathrm{Net}(\mathbf{x})$ and the faulty network $\mathrm{Net}_f(\mathbf{x})$. Our objective differs from such related works, which only try to maximize the output deviation to measure the presence of fault [45], [56]. However, the faults that do not lead to misclassification can consider negligible.

We thus aim at changing the index with the maximum network activation in the presence of a fault. According to [80], this be achieved by increasing the deviation in the Softmax distribution of the network between the faulty network $\mathrm{Net}_f(\mathbf{x})$ and fault free network $\mathrm{Net}(\mathbf{x})$. Consequently, a test input (test pattern) $\mathbf{x}^\star$ is generated as

$$\mathbf{x}^\star = \underset{\mathbf{x}\in\mathbb{X}}{\mathrm{argmax}}\ o(\mathbf{x}, f) \tag{7.5}$$

with

$$o(\mathbf{x}, f) = \left\| \mathrm{Softmax}\left(\mathrm{Net}(\mathbf{x})\right) - \mathrm{Softmax}\left(\mathrm{Net}_f(\mathbf{x})\right) \right\|_2^2,$$

where $\|\cdot\|_2^2$ denotes the squared euclidean norm, and the set $\mathbb{X}$ denotes the set of feasible input values. As an example, for images, only pixel values in $[0, 255]$ may be allowed. Such constraints can be addressed via element-wise value clipping. Due to the differentiability of the objective function $o(\mathbf{x}, f)$ with respect to $\mathbf{x}$ in Eq. (7.5), the optimisation can be performed efficiently via gradient-based optimizes, e.g., Adam [27]. However, multiple restarts may be required to obtain a good solution due to the complicated optimisation objective involving the NN.

It is important to stress that test pattern generation (using logits) occurs in software with full network access, including the Softmax output. Conversely, during in-field operation (inference mode), the network may be deployed on a sealed accelerator where only the input and class output are observable. Our method, primarily used for In-System Tests (ISTs) as a functional testing approach, does not assume access to logits in inference mode, as it aims to alter the class output, which distinguishes it from offline (e.g. manufacturing) test and structural testing

7.5.3. Compressed test pattern generation

In case of stuck-at faults, a large amount of test patterns may be extracted. To reduce the required storage costs *for each* test pattern $\mathbf{x}$, we will directly generate them in a compressed representation. For this, we first generate a set of basis vectors $\mathbf{B}$ (shared by all test patterns), and then represent each test pattern only through its a coefficient vector. In other words, a test pattern $\mathbf{x}_i = \mathbf{B}\boldsymbol{\alpha}_i$. Note that, e.g. in case of the popular image data set $\mathbf{x} \in \mathbb{R}^{32\times32\times3}$ (flattened to a vector), while $\boldsymbol{\alpha} \in \mathbb{R}^D$ and $\mathbf{B} \in \mathbb{R}^{(32\times32\times3)\times D}$. Hence, for a given basis $\mathbf{B}$, we generate our test patterns as

$$\boldsymbol{\alpha}^\star = \underset{\boldsymbol{\alpha}\in\mathbb{R}^d}{\operatorname{argmax}}\ o(\mathbf{B}\boldsymbol{\alpha}, f) \quad \text{s.t.} \quad \mathbf{B}\boldsymbol{\alpha} \in \mathbb{X} \tag{7.6}$$

Thus, through this formulation, we implicitly search for test patterns $\mathbf{x}$ that lie in the subspace spanned by $\mathbf{B}$.

Note that searching in the component space (the space of the $\boldsymbol{\alpha}$), introduces an additional constraint. Namely, we need to make sure, that the resulting $\mathbf{x}$ will be feasible, i.e., $\mathbf{x} = \mathbf{B}\boldsymbol{\alpha} \in \mathbb{X}$. In the following, we describe a procedure to address simple box-constraints to ensure, e.g. valid input pixels in $[0, 255]$. For more complicated $\mathbb{X}$, a more involved treatment may be required.

To ensure box-constraints for $\mathbf{x}$, like $x_i \in [0, 1]$, we leverage the following normalisation operation

$$\text{Normalize}(\mathbf{x}) := (\mathbf{x} - \mathbf{1} \cdot \min_i\{x_i\})(\max_i\{x_i\} - \min_i\{x_i\})^{-1},$$

where $\mathbf{1}$ denotes a vector of all ones of appropriate size and $\min_i\{x_i\}$ (respectively max) denote the value of the minimum (maximum) entry of $\mathbf{x}$. Equivalently, the normalisation can be reformulated to map to any bounded interval $[x_{\min}, x_{\max}]$ by adding $x_{\min}$ and multiplying by $(x_{\max} - x_{\min})$.

In most cases, i.e. if $\mathbf{x}$ has at least two different entries, the normalisation operation is defined. We now reformulate Eq. (7.6) unconstrained as

$$\boldsymbol{\alpha}^\star = \underset{\boldsymbol{\alpha}\in\mathbb{R}^d}{\operatorname{argmax}}\ o\left(\text{Normalize}(\mathbf{B}\boldsymbol{\alpha}), f\right). \tag{7.7}$$

As $\text{Normalize}(\mathbf{x})$ is differentiable (almost everywhere) and allows for the computation of gradients, the new optimisation objective still qualifies for gradient-based optimisation.

7.5.4. Choosing and generating the basis vector

What remains to be defined is the choice of the column vectors $\mathbf{b}_1, \cdots, \mathbf{b}_D$ of $\mathbf{B}$ which span the space for $\mathbf{x}$. For computational reasons, choosing $\mathbf{B}$ as an orthogonal matrix may be beneficial. However, for

simplicity, we simply generate the vectors $\mathbf{B}$ pseudo-randomly with fixed random seeds (e.g., integers from 1 to D). This has the advantage that we do not need to store $\mathbf{B}$, as its columns $\mathbf{b}_k$ can be generated reproducibly on-demand. More specifically, the calculation of $\mathbf{x} = \mathbf{B}\boldsymbol{\alpha} = \sum_k \mathbf{b}_k \cdot \alpha_k$, can, aside form the storage required for the $\boldsymbol{\alpha}$, be done on the fly.

Note that the quality of the basis vectors influences the compression ratio, and an improved basis may lead to a lower number of components needed. Unfortunately, if those improved basis vectors could not be generated on demand (as in our approach), they would have to be stored, which may mitigate storage gains. Nevertheless, trying to find improved basis vectors could still be worth it as the number of coefficient vectors to be stored depends on the number of faults to be detected, while the number of basis vectors remains constant (yet possibly large). Unfortunately, integrating the basis vectors into the search process is not straightforward. In contrast to the coefficients $\boldsymbol{\alpha}$, which are independent between test patterns, the basis $\mathbf{B}$ is shared by all test patterns. In other words, we could not solve Eq. (7.7) independently for each fault but would have to solve a large problem involving $\mathbf{B}$ and $\boldsymbol{\alpha}^{(n)}, n = 1, \cdots, N$ for all faults from a fault list $\mathcal{F} := \{f^{(1)}, \cdots, f^{(n)}, \cdots, f^{(N)}\}$ simultaneously.

To summarize the procedure for compressed test pattern generation: Given a fault list $f \in \mathcal{F}$ we first choose a random seed for the generation of $\mathbf{B}$, We then solve Eq. (7.7), and store the coefficients $\boldsymbol{\alpha}^{(n)}$ for the n-th fault. For testing, we reconstruct the test pattern $\mathbf{x}^{(n)}$ for the n-th fault as Normalize($\mathbf{B}\boldsymbol{\alpha}^{(n)}$). The pseudo code for the test pattern generation algorithm can be seen in Algorithm 6 and an illustration of the objective can be seen in Fig 7.6.

Algorithm 6 The test pattern generation algorithm. The set of coefficients of the test patterns is stored in $\mathcal{A}$. The n-th test patterns can be retrieved on-demand as $\mathbf{x}^{(n)} = \text{Normalize}(\mathbf{B}\boldsymbol{\alpha}^{(n)})$. The basis vectors $\mathbf{B}$ are generated.

```
# Set design parameters
Choose d (number of basis vectors) to generate B
# Generate list of faults that should be detected
F := {f^(1), ..., f^(n), ..., f^(N)}
A ← ∅
for f in F do
    α* ← argmax_{α ∈ R^d} o(Normalize(Bα), f)
    A ← A ∪ {α*}
end for
```

To further illustrate our approach, N test vectors (corresponding to N faults or even after fault collapsing) are generated. Thus, N test patterns need to be stored.

So if each input image has P pixels, $N \times P$ pixels are needed to be stored, and given that each pixel requires t bits, $N \times P \times t$ bits are needed to represent and store all test patterns. Now, with our proposed method, if we have D basis, each test pattern is represented by D coefficients (real numbers). Also, the basis images do not need to be explicitly stored since $D << P$, we achieve a huge saving in the storage. For instance, if we have 1000 test patterns, image size 28×28, 20 basis vector, and the test pattern is stored using 32 floating point precision, so 4 bytes, then the storage cost for optimizing the whole test pattern is 4×(784×1000) equal to 313,6000 bytes. However, if the patterns were optimized with a joint basis, the storage cost would be 4× (1000×20), equivalent to 80,000 bytes, giving a compression ratio of 39.2×.

Finally, it is important to stress that compaction methods, i.e., the removal of patterns that detect covered faults, e.g. based on fault collapsing may help to further decrease storage costs. Such compaction approaches are orthogonal to our work and may be freely combined with the presented approach.

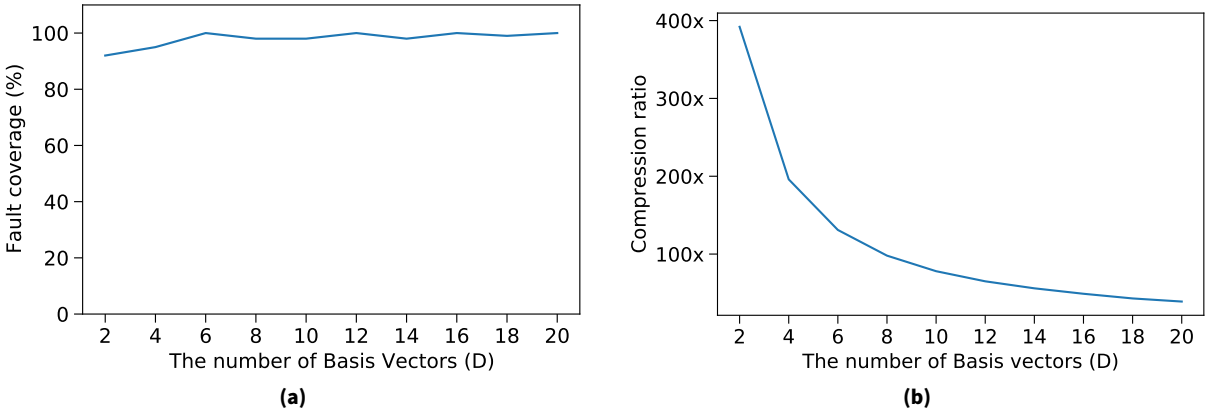

Figure 7.7.: An illustration for the fault coverage with respect to to the number of basis vectors D (a), the compression ratio with respect to the number of basis vectors D (b) for MLP (MNIST) model using 20 test patterns, ReLU and Stuck-at middle value.

7.6. Experimental Evaluation

In this section, the experimental setup is described, followed by a comparison between the storage required to store full test patterns and the proposed basis-driven generation approach. The results are then presented and discussed.

7.6.1. Datasets and Experimental Setup

The experiments were performed using 32-bit floating-point (FP32) weights and implemented in *PyTorch* [49] on the MNIST [24] and CIFAR-10 [25] datasets.

An MLP and LeNet-5 were used for MNIST, while ConvNet-7 was used for CIFAR-10. Additionally, to demonstrate the scalability of the proposed approach for larger models, VGG-19 [30] was employed on CIFAR-10.

The learning rate was determined using a grid search on a logarithmic scale from 10^{-7} to 1. The MLP experiments were conducted on an Intel(R) Xeon(R) CPU (2.0 GHz, 12 GB RAM), while the CNN experiments were performed on an NVIDIA GP100-892-A1 (Pascal) GPU with 16 GB memory.

7.6.2. Experimental Results and Discussion

Referring to Section 7.5.1, single and multiple SAFs were injected at the outputs of neurons in MLP, LeNet-5, ConvNet-7, and VGG-19 models across different layers. The accuracy reached 96% on MNIST and LeNet-5, and 94% on CIFAR-10, prior to fault injection. In the experiments, ReLU6 was set to its maximum value (6) and minimum value (0), while Sigmoid was set to its maximum (1) and minimum (0).

To justify the relevance of the neuron-level SAF model, fault simulations were first performed by injecting multiple bit-level faults into weights. These faulty weights were then propagated through the neuron, and their effect on the neuron output was analyzed.

In these simulations, different percentages of weights were injected with varying percentages of bit flips. It was observed that injecting less than 80% of the weights with 90% to 100% bit flips did not change the neuron output; therefore, these faults were masked (i.e., redundant). However, injecting more than 80% of the weights with 90% to 100% bit flips resulted in observable changes in the neuron output, indicating that the fault impact becomes visible.

Table 7.2 shows the effect of injecting 100% of the weights with 100% bit flips. Sigmoid and ReLU6 activation functions were selected for illustration due to their bounded output ranges. The values before and after fault injection are presented in Table 7.2.

Table 7.2.: The fault simulation shows the effect of injecting weight faults at the bit level and its impact on the neuron output using Sigmoid and ReLU6 activations. Here, we injected 100% of the weight with 100% bit flips.

Activation function	Before fault injection (fault-free value)	After fault injection (faulty value)
Sigmoid	1.00	0
	0.99	0
	0.72	1
	0.66	1
	0.35	1
	0.42	0
	0.18	1
	0.00	1
ReLU6	6.00	0
	5.08	6
	4.77	0
	3.39	0
	1.90	0
	0.43	6
	0.00	6

It can be observed that the neuron outputs after fault injection tend toward the minimum or maximum bounds of the activation function. This behavior is equivalent to a neuron stuck-at a deterministic value, which validates the effectiveness of the neuron-level SAF model. These experiments confirm that multiple weight faults either remain masked or result in neuron outputs being driven to extreme activation values. Consequently, the neuron fault model can be safely used as an abstraction of weight faults while being computationally more efficient.

7.6.2.1. Compressed Test Pattern Generation Results

Based on the above analysis, single or multiple stuck-at faults were injected at neuron outputs. Faults that caused significant degradation in inference accuracy were selected for test pattern generation.

It is observed that the impact of faults depends on their location, with earlier layers generally having a greater effect than layers closer to the output.

After fault injection, test patterns were generated such that each pattern corresponds to one or multiple injected neurons. Fault detection was determined by observing misclassification between the outputs of the fault-free and faulty networks.

In the output layer, each neuron corresponds to a class, and its output represents the confidence for that class. The neuron with the highest confidence determines the predicted class. For each injected neuron n_j, the confidence scores were computed for both the fault-free and faulty networks. A fault was considered detected if the predicted class labels differed. The generated test patterns and experimental settings are reported in Table 7.3.

The bar chart in Fig. 7.8 illustrates how the proposed test patterns detect faults in the most vulnerable neurons. For example, for neuron seven, the predicted label changes from 8 (fault-free) to 3 (faulty), indicating successful fault detection.

For the proposed compression approach, the number of basis vectors was selected based on achieving high fault coverage (above 90%). As shown in Table 7.3, increasing the number of basis vectors generally leads to higher fault coverage, as a higher-dimensional subspace can be explored.

Table 7.3.: The overall experimental results on different models and datasets, Uncompressed refers to the test patterns in their uncompressed representation, Compressed refers to the compressed representation of the test pattern with joint basis. In fault modeling, we use single and multiple stuck-at faults, and we stuck the output of the neuron. The percentage next to neuron fault (SAm) refers to the percentage of the fault injected neurons in the multiple neuron fault injection scenario. If no percentage is specified, it implies that a single neuron is injected.

Experimental setup					Test pattern approaches					
Model and Dataset	Activation function	Fault model	# Basis vectors (D)	Test patterns	Rand.	Adv. [45]	Proposed			
					Fault coverage (%)			Storage cost (bytes)		Compression ratio
								Uncompressed	Compressed (Proposed)	
MLP (MNIST)	Sigmoid	SA0	30	20	50	10	98	62,720	2400	26.1×
			25		52	10	95		2000	31.4×
			20		55	10	98		1600	39.2×
			35	60	5	0	92	188,160	8400	22.4×
			30		1	0	90		7200	26.1×
			25		3	0	88		6000	31.4×
		SA0(10%)	30	20	50	0	100	62,720	2400	26.1×
		SA0(15%)	25		51	0	100		2000	31.4×
		SA1	20		10	0	100		1600	39.2×
			15		10	0	100		1200	52.3×
	Tanh	SA1	20		40	20	92		1600	39.2×
			15		44	20	89		1200	52.3×
		SA0	20		49	15	92		1600	39.2×
			15		40	25	88		1200	52.3×
	ReLU6	SA6	20		28	15	95		1600	39.2×
			15		39	15	94		1200	52.3×
		10			34	15	95		800	78.4×
		SA6(10%)			75	0	100		800	78.4×
		SA0	20		10	0	100		1600	39.2×
			15		30	0	95		1200	52.3×
LeNet-5 (MNIST)	ReLU6	SA6	20		40	5	95		1600	39.2×
			15		15	10	95		1200	52.3×
		10			10	5	98		800	78.4×
		SA6(10%)			5	0	100		800	78.4×
		SA6(15%)		60	50	0	100	188,160	2400	78.4×
		SA0(15%)			10	0	100		2400	78.4×
		SA0(10%)		20	0	0	100	62,720	800	78.4×
		SA0	20		20	15	96		1600	39.2×
			15		0	10	92		1200	52.3×
	Sigmoid	SA0	20		0	0	90		1600	39.2×
			15		0	5	95		1200	52.3×
			10		3	5	100		800	78.4×
ConvNet-7 (CIFAR-10)	ReLU6	SA6	25		30	15	95	245,760	2000	122.9×
			20		20	10	92		1600	153.6×
		10			50	25	95		800	307.2×
		SA6(10%)			70	0	100		800	307.2×
		SA0	30		20	5	95		2400	102.4×
			25		25	5	92		2000	122.9×
		SA6	30	30	63	20	93	368,640	3600	102.4×
		25			13	37	95		3000	122.9×
		SA6(10%)			0	0	100		3000	122.9×
				60	0	0	100	737,280	6000	122.9×
			20		20	5	97		4800	153.6×
VGG-19 (CIFAR-10)	ReLU	SA0(5%)	50	4096	0	0	100	50,331,648	819200	61.4×
			40		0	0	100		655360	76.8×
		SA6(5%)	50		0	0	100		819200	61.4×
			40		0	0	99		655360	76.8×

However, in some cases (e.g., LeNet-5 with sigmoid activation and 10 basis vectors), better coverage was achieved with fewer basis vectors. This may be attributed to the non-convex nature of the optimization problem, where the global optimum is not always reached. Restarting the optimization procedure may improve results.

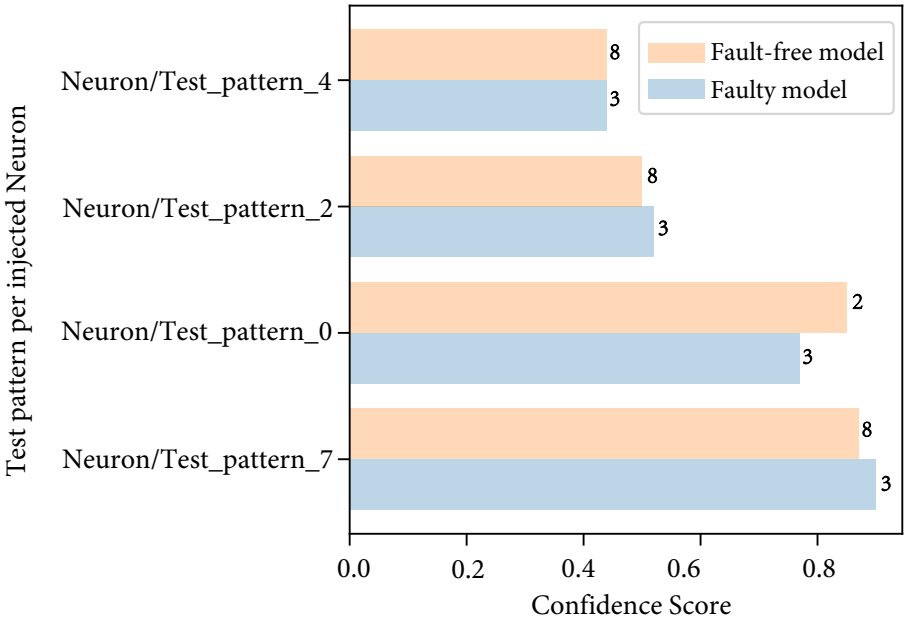

Figure 7.8.: Label misclassification illustration.

It is also observed that for multiple fault injections (e.g., SA0 (10%), SA6 (15%)), the proposed method achieves up to 100% fault coverage. This demonstrates the scalability of the approach, even for large models such as VGG-19.

Additionally, the proposed method was compared with adversarial [45] and randomly generated test patterns. As shown in Table 7.3, the proposed approach consistently achieves higher fault coverage, reaching up to 100% for both single and multiple faults.

Regarding storage, test patterns are represented using 32-bit floating-point precision (4 bytes). Storing raw test patterns requires $4 \times (N \times P) \times C$ bytes, where N is the number of patterns, P is the number of pixels, and C is the number of channels.

In contrast, the proposed approach stores only the coefficient vectors $\boldsymbol{\alpha}$, resulting in a storage requirement of $D \times N$. Since the basis $\mathbf{B}$ can be generated on demand, it does not need to be stored.

As shown in Table 7.3, this results in significantly reduced storage requirements. For example, for CIFAR-10, a compression ratio of up to 307.2× is achieved, demonstrating strong scalability.

Finally, to evaluate the impact of different basis initializations, 10 different random seeds were used. The best fault coverage across these seeds was reported. Fig. 7.7-a shows that increasing the number of basis vectors improves coverage, while Fig. 7.7-b illustrates the trade-off between coverage and compression. For example, for MLP, $D = 6$ achieves a good balance (96.7% coverage and 131× compression).

7.7. Chapter Summary

This chapter addressed the challenge of storage-efficient functional testing for DNN accelerators operating in resource-constrained and safety-critical environments. While previous chapters focused on maximizing fault coverage and minimizing test-set size, the methods presented here explicitly targeted the memory overhead associated with storing and deploying test patterns in-field.

Two storage-aware testing strategies were introduced. First, the NBIST framework enables on-demand test generation through a lightweight generator network co-deployed with the NUT. Instead of storing full-resolution test inputs, NBIST stores only the generator parameters, a seed value for a pseudo-random number generator, and the expected output labels. During testing, pseudo-random vectors are mapped

to structured test inputs, which are then evaluated by the NUT and compared against the stored labels to determine the presence of faults. Across multiple models, datasets, and composite fault scenarios, NBIST achieved up to 100% fault coverage while requiring significantly fewer stored bytes than raw or adversarial test sets. By eliminating the need to store complete input patterns, NBIST substantially reduces memory overhead and enables fully autonomous in-field validation without additional hardware modifications.

Second, a basis-driven compressed test generation framework was proposed. In this approach, test patterns are represented as linear combinations of basis vectors. Instead of storing entire input tensors, only a small set of coefficients per test pattern is retained, while the shared basis vectors are generated reproducibly on demand. The method optimizes test patterns by maximizing the Softmax deviation between fault-free and faulty networks, thereby encouraging misclassification under neuron stuck-at faults. The experimental results demonstrated that the compressed representation achieved up to 100% fault coverage and reduced storage requirements by up to 307.2× compared to direct storage of test inputs. Furthermore, experimental validation confirmed that multiple weight faults can be effectively abstracted as neuron SAFs, supporting the modeling assumptions underlying the method.

Together, the two approaches explore different philosophies for storage-efficient test deployment. NBIST uses nonlinear learned generation for flexible, adaptive test creation, while the basis-driven method relies on structured linear representations for analytical compression and deterministic reconstruction. Both frameworks integrate storage considerations directly into the test generation process rather than relying on post-generation compaction.

8. Unified framework for in-field testing.

8.1. Chapter Overview

In this chapter, a unified framework is presented for in-field functional testing of DNNs deployed on specialized AI-HAs. As DNNs continue to serve as the backbone of modern AI systems across domains such as healthcare, finance, autonomous driving, and industrial automation [69], [70], ensuring their reliable operation becomes increasingly critical—particularly in safety-sensitive environments.

Existing DNN fault detection techniques often rely on predefined fault lists [90], which limits their ability to generalize to unforeseen or probabilistic hardware failures. Other approaches generate large sets of test patterns and subsequently apply compaction techniques [63], [80] to satisfy storage constraints. However, such post-processing strategies do not inherently consider storage limitations during test synthesis. In addition, several prior works focus primarily on adversarial-style input perturbations [45] or on specific fault types, highlighting the need for a more generalized and deployment-aware testing methodology.

To address these challenges, this chapter introduces a unified test pattern generation approach that explicitly incorporates storage and execution constraints during test synthesis. The framework generates a fixed-size set of test patterns jointly and aims to maximize expected fault coverage under a probabilistic fault model. By integrating resource constraints directly into the generation process, the resulting test patterns satisfies practical in-field deployment requirements while maintaining high fault coverage.

The key contributions of this chapter are summarized as follows:

- A fault-type-agnostic functional testing methodology adaptable to probabilistic fault models.
- A storage-aware test generation framework that integrates memory and execution constraints directly into the synthesis process, thereby eliminating the need for post-generation compaction.
- Experimental validation across multiple fault types, models, and datasets, demonstrating fault coverage reaching up to 100%.

This chapter consolidates the concepts developed throughout the thesis into a coherent in-field testing perspective, providing a scalable and deployment-oriented solution for reliable functional validation of DNNs-based AI accelerators.

The remainder of this chapter is organized as follows. Section 8.2 introduces the fault model used for in-field functional testing and presents the unified optimization formulation that incorporates coverage, storage, and execution constraints. Section 8.3 presents the experimental evaluation and analysis. Finally, Section 8.4 concludes the chapter with a discussion of practical deployment implications and future research directions.

8.2. Storage-Aware Test Pattern Generation Under Probabilistic Fault Models

In this section, a unified functional test pattern generation approach is proposed. the method designed to ensure the functional reliability of DNNs deployed on AI-HAs. The method directly incorporates practical constraints, such as limited memory for storing test patterns. Using a probabilistic fault model, the approach captures a wide range of hardware-induced faults across different components, including memory cells, combinational logic, and sequential elements.

8.2.1. Probabilistic Fault Modeling

Following the abstraction in Section 2.3, hardware faults are modeled as transformations

$$f : \text{Net} \mapsto \text{Net}_f,$$

where an error occurs if $\text{Net}(\mathbf{x}) \neq \text{Net}_f(\mathbf{x})$ for a given input $\mathbf{x}$. Concrete fault instances are sampled as $f^{(n)} \sim p(f)$.

We consider parameter-level fault mechanisms, since weights dominate both memory footprint and computation:

Weight perturbation transformations. Continuous parameter deviations are modeled by additive or multiplicative noise applied to θ, capturing hardware variations and analog drift.

Bit flip transformations. Transient faults are modeled by randomly flipping bits in the binary representation of θ with probability ρ, representing soft errors in memory or datapaths.

Stuck-at transformations. Permanent defects are modeled by forcing a fraction ρ of parameters to a fixed value v, capturing persistent memory corruption.

Composite transformations. Multiple fault mechanisms are combined via composition (e.g., $f_{\text{perturb}} \circ f_{\text{bitflip}}$) to represent realistic mixed-fault scenarios. Fault lists can be interpreted as discrete distributions over such transformations within the same probabilistic framework.

8.2.2. Test Pattern Generation

In the following, a unified test pattern generation approach is outlined. The main characteristics considered are as follows:

1. The generation of test patterns may use white-box information, but testing should be considered as a black-box process.
2. The test pattern should be generated in consideration of the fault distribution $p(f)$.
3. Storage constraints for test patterns should be accounted for at the test pattern generation time.

In real-world deployments, DNNs often operate in black-box environments, where only the input $\mathbf{x}$ and the corresponding classification output $\mathbf{y}$ are accessible. This is particularly relevant in system-level test and in-field testing scenarios, where internal computations are not directly observable. Therefore, an effective test pattern generation strategy should aim to induce output discrepancies in the presence of faults, allowing faults to be detected solely through changes in the observable output.

Moreover, the generation process should consider the underlying fault distribution and adhere to predefined storage constraints. Specifically, the size of the generated test set must be optimized to fit within the available on-chip BIST memory budget while maintaining high fault detection effectiveness.

To address the first characteristic, we draw inspiration from previous works [80], [90], [92], where an optimization objective is defined to maximize a difference $d(\mathbf{x}, f)$ in the predicted class distributions for a given test pattern $\mathbf{x}$ in the presence of a fault f.

One such objective is given by

$$d(\mathbf{x}, f) = \| \text{Softmax}(\text{Net}(\mathbf{x})) - \text{Softmax}(\text{Net}_f(\mathbf{x})) \|_2^2, \tag{8.1}$$

where $\mathrm{Net}(\mathbf{x})$ denote the fault-free network, and $\mathrm{Net}_f(\mathbf{x})$ denotes the faulty network [80], [90].

Alternatively, as described in [92], the Kullback-Leibler divergence

$$d(\mathbf{x}, f) = \sum_c \mathrm{Softmax}(\mathrm{Net}(\mathbf{x}))_c \ln\left(\frac{\mathrm{Softmax}(\mathrm{Net}(\mathbf{x}))_c}{\mathrm{Softmax}(\mathrm{Net}_f(\mathbf{x}))_c}\right)$$

between the softmax distributions of $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}_f(\mathbf{x})$ can be used.

Regardless of the specific function $d(\mathbf{x}, f)$, maximizing such an objective function should yield test patterns $\mathbf{x}$ that are suitable for black-box testing, as they encourage different predicted classes in the presence of a fault. Consequently, many previous works, see [45], [55], [80], [90], [92] generate test patterns this way. Unfortunately, such approaches often only consider single faults when generating a test pattern. While the generated test patterns may also cover multiple faults by chance, they are usually not generated with this idea in mind.

Moreover, generating test patterns based on individual faults may fail to accurately represent the overall fault distribution $p(f)$.

To address these issues, we seek to find the entire set of test patterns $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k, \ldots, \mathbf{x}_K\}$ jointly with the aim of achieving high fault coverage in expectation according to $p(f)$.

The optimization objective for this can be formalized by

$$\begin{aligned}\mathcal{X}^\star &= \operatorname*{argmax}_{\mathcal{X}} \quad \mathbb{E}_{p(f)}\left[\max_{\mathbf{x}\in\mathcal{X}} d(\mathbf{x}, f)\right] \\ &= \operatorname*{argmax}_{\mathcal{X}} \int \left(\max_{\mathbf{x}\in\mathcal{X}} d(\mathbf{x}, f)\right) p(f)\,\mathrm{d}\,f,\end{aligned} \tag{8.2}$$

where $\mathcal{X}^\star$ characterizes the optimal set of test patterns.

Through the inner max operation, only one of the test patterns, specifically the one which leads to the highest difference $d(\mathbf{x}, f)$, is made responsible for the detection of the fault, while other test patterns remain free to focus on other faults instead. Consequently, this objective encourages the test patterns in $\mathcal{X}$ to distribute the effort to detect the different faults caused by $p(f)$ among each other. Furthermore, faults that lead to similar/equivalent output behavior are naturally captured by the same test pattern $\mathbf{x}$ (for which $d(x, f)$ is the largest). Finally, since the size of the set $\mathcal{X}$ is specified in the objective (argument), the upper bound of the storage limitation is explicitly taken into account. The objective can usually not be calculated analytically; however, since it is an expected value, it can be estimated using Monte Carlo techniques for given $\mathcal{X}$ with sampled faults $f^{(n)} \sim p(f)$. The samples $f^{(n)}$ can be obtained from any probabilistic fault model, which allows for sampling or is defined through a sampling procedure (e.g., fault injection). Hence, an estimate can be obtained by

$$\mathbb{E}_{p(f)}\left[\max_{\mathbf{x}\in\mathcal{X}} d(\mathbf{x}, f)\right] \approx \frac{1}{N}\sum_{n=1}^{N} \max_{x\in\mathbb{X}^\star} d(x, f^{(n)}) \quad \text{with} \quad f^{(n)} \sim p(f)$$

To find a suitable set of test patterns $\mathcal{X}$ for a given network $\mathrm{Net}(\cdot)$ and a probabilistic fault model $p(f)$, we perform gradient-based optimization on the objective function in (8.2). The gradients of the objective are given by

$$\begin{aligned}\nabla_{\mathcal{X}} \mathbb{E}_{p(f)} \left[\max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f) \right] &= \nabla_{\mathcal{X}} \int \left(\max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f) \right) p(f) \, \mathrm{d} f \\ &= \int \left(\nabla_{\mathcal{X}} \max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f) \right) p(f) \, \mathrm{d} f \\ &= \mathbb{E}_{p(f)} \left[\nabla_{\mathcal{X}} \max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f) \right]\end{aligned}$$

Here, the gradient and the integral operator can be exchanged through the assumption that the faults are not influenced by the test patterns used. This assumption should mostly hold in practice. In this reformulation, the gradient can also be written as an expected value over $p(f)$ and similar to the objective, the gradient can be estimated using Monte Carlo estimation, i.e.,

$$\mathbb{E}_{p(f)} \left[\nabla_{\mathcal{X}} \max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f) \right] \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_{\mathcal{X}} \max_{x \in \mathcal{X}} d(x, f^{(n)}) \quad \text{with} \quad f^{(n)} \sim p(f). \tag{8.3}$$

The gradients calculated this way can then be used together with any gradient-based optimization algorithm, e.g., gradient ascent. However, due to the stochastic nature of the gradient estimates, we would opt for algorithms designed with stochastic objectives in mind, such as Adam [27]. It should also be noted that the max operation, even though not smooth, does not pose a substantial problem for gradient computation. The function is still differentiable almost everywhere and gradients can be obtained using common automatic differentiation frameworks such as *pytorch* [49]. As usual, the gradient-based optimization procedure can be started from randomly initialized $\mathbf{x}_k$, $k = 1, \ldots, K$ which are then iteratively updated. To ensure a high-quality solution, it can also be restarted and rerun several times to compensate for unfortunate initializations.

Depending on the input domain of the network, the test patterns $\mathbf{x} \in \mathcal{X}$ must be in a specified domain. Such constraints on $\mathcal{X}$ can be addressed via projections of the iterates. For example, in the case of images, $\mathbf{x}$ have to be element-wise in $[0, 255]$. Here, the projection simply relates to element-wise clipping to the specified range.

When estimating gradients according to (8.3), the number of fault samples for each step of the algorithm influences the quality of the estimate. Consequently, aside from a constant number of samples in each step, also a varying number of samples can be used. For example, in the beginning of the optimization, lower quality (higher variance) estimates using fewer samples can be used, while the number of samples can be increased towards the end of the optimization procedure to obtain higher quality (lower variance) estimates of the gradients.

Due to the stochastic nature and complexity of the optimization, convergence of the procedure is unlikely. Thus, classical stopping criteria such as small gradient norms cannot be used to terminate the procedure. We therefore suggest to run the algorithm for a predefined number of steps T, or until no improvement has been observed for a predefined number of steps while tracking the best result achieved so far. For each step, the same samples $f^{(n)} \sim p(f)$ can be used to estimate the objective and the gradient. Algorithm 7 describes the procedure of finding patterns $\mathcal{X}^\star$ via gradient ascent and is illustrated in Fig. 8.1.

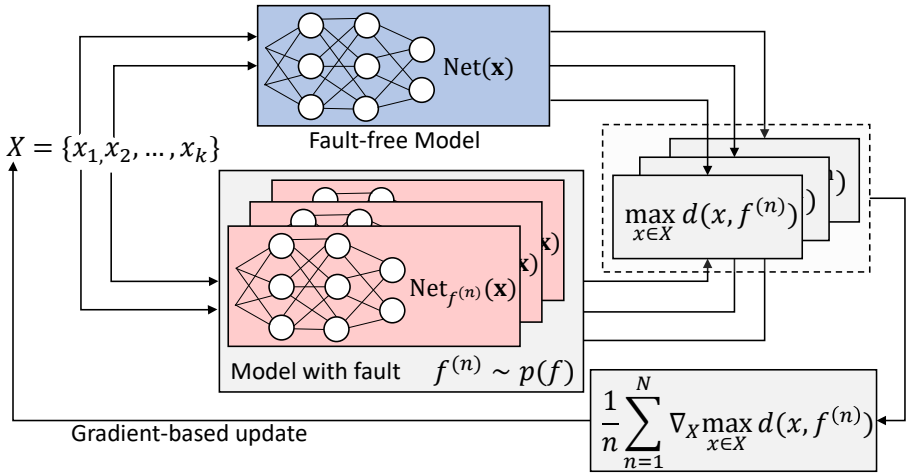

Figure 8.1.: An illustration of the proposed approach. In each step, the gradient of the objective $d(\mathbf{x}, f)$ is computed using sampled faults from $p(f)$.

Algorithm 7 Test Pattern Generation via Gradient Ascent

Inputs: NN$(\cdot)$ (network model), $p(f)$ (fault distribution), K (test set size), α (gradient step size), N (Monte Carlo samples), T (total iterations).

procedure GenTestPattern(NN$(\cdot)$, $p(f)$, K, α, N, T)
    # Initialize variables
    $\mathcal{X} \leftarrow \{\mathbf{x}_k\}_{k=1}^{K}$ where $\mathbf{x}_k \sim \mathcal{N}(0, 1)$
    $\mathcal{X}^\star \leftarrow \mathcal{X}$ ▷ Set of best test patterns
    $o^\star \leftarrow -\infty$ ▷ Best objective value found so far
    for t in $\{1, \ldots, T\}$ **do**
        # Estimate o and $\mathcal{G}$ with shared samples
        Sample $\{f^{(n)}\}_{n=1}^{N} \sim p(f)$
        $o \leftarrow \frac{1}{N} \sum_{n=1}^{N} \max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f^{(n)})$
        $\mathcal{G} \leftarrow \frac{1}{N} \sum_{n=1}^{N} \nabla_{\mathcal{X}} \max_{\mathbf{x} \in \mathcal{X}} d(\mathbf{x}, f^{(n)})$
        # Update test patterns via gradient ascent
        $\mathcal{X} \leftarrow \mathcal{X} + \alpha \cdot \mathcal{G}$
        $\mathcal{X} \leftarrow \text{Proj}[\mathcal{X}]$ ▷ Project to ensure feasibility
        if $o > o^\star$ **then**
            $\mathcal{X}^\star \leftarrow \mathcal{X}$
            $o^\star \leftarrow o$
        end if
    end for
    return $\mathcal{X}^\star$
end procedure

8.3. Experiments

8.3.1. Datasets and Experimental Setup

This chapter utilizes both MLP and CNN models for classification tasks, as well as a segmentation model for pixel-wise classification. For classification, the CIFAR-10 [25] and MNIST [24] datasets are used to

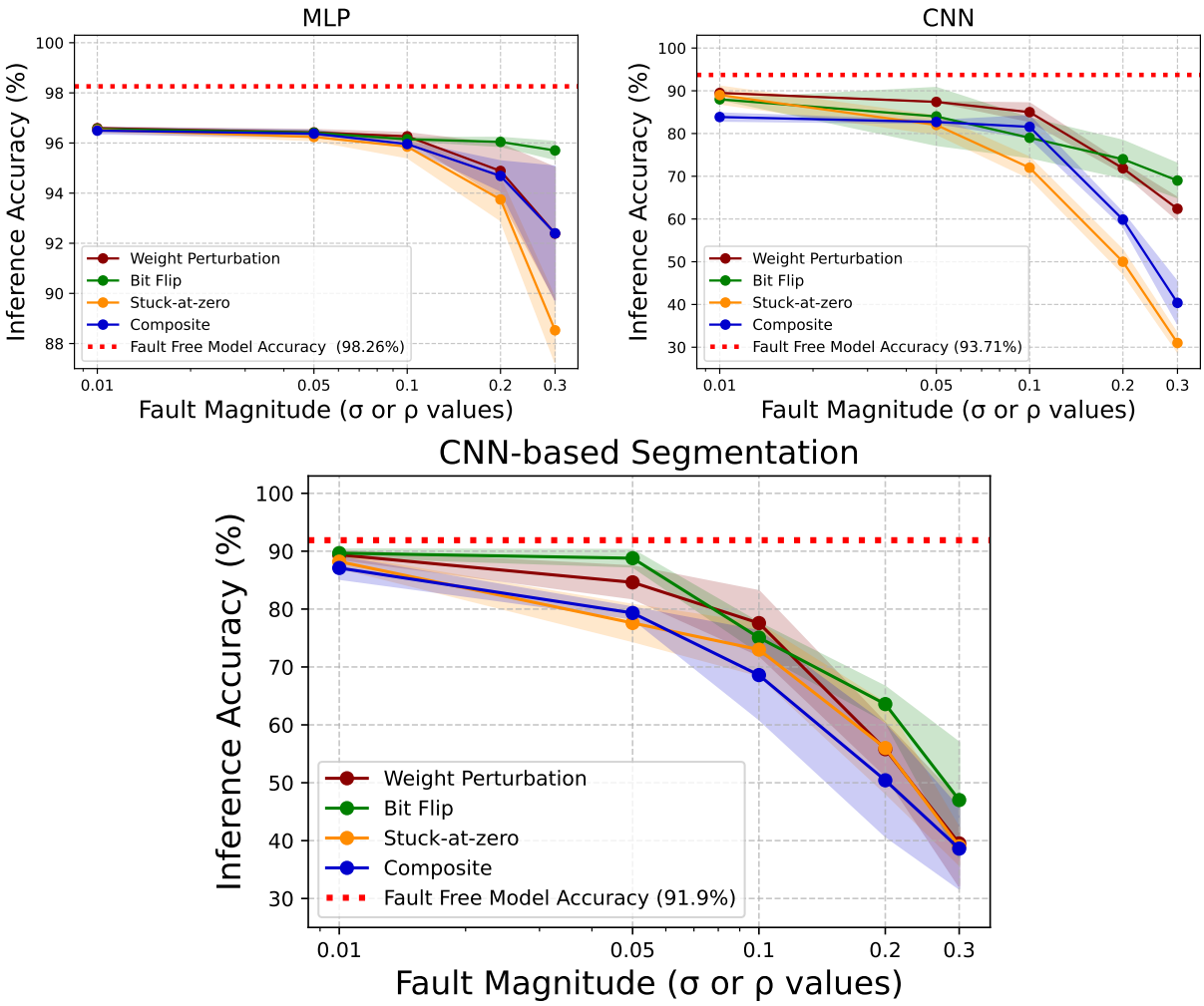

Figure 8.2.: Impact of Different Fault Injection on Inference Accuracy Across Different Models

evaluate the models. CIFAR-10 consists of 60,000 color images (32×32 pixels) across 10 categories, including airplanes, automobiles, birds, cats, and more, making it a benchmark for object recognition tasks. MNIST, on the other hand, is a simpler dataset containing 70,000 grayscale images (28×28 pixels) of handwritten digits (0-9), widely used for evaluating fundamental classification algorithms.

The architecture of the MLP is a three-fully connected network of size 20. The input images are flattened into vectors and passed through multiple dense layers with ReLU activation functions. In the CNN we use a custom 7-layer NN (ConvNet-7) with four convolutional layers and three fully connected layers. Additionally, to evaluate the scalability of our approach, we also perform experiments with Visual Geometry Group (VGG-19) [30]. VGG-19 is a deep CNN consisting of 19 layers (16 convolutional layers, 3 fully connected layers, 5 Max Pool layers).

For segmentation, a CNN-based segmentation model is used. To evaluate the proposed method on segmentation tasks, we use a custom CNN-based segmentation model with a total of 10 layers. The architecture includes 8 convolutional layers with ReLU activations and 3 max-pooling layers for down-sampling. A final 1×1 convolution layer maps the features to the desired number of classes, followed by bilinear up-sampling to recover the original image resolution for pixel-wise classification. The segmentation task is evaluated using the CamVid [18] dataset. CamVid consists of RGB images of road scenes along with pixel-wise annotations for different object classes, such as roads, cars, pedestrians, and sky.

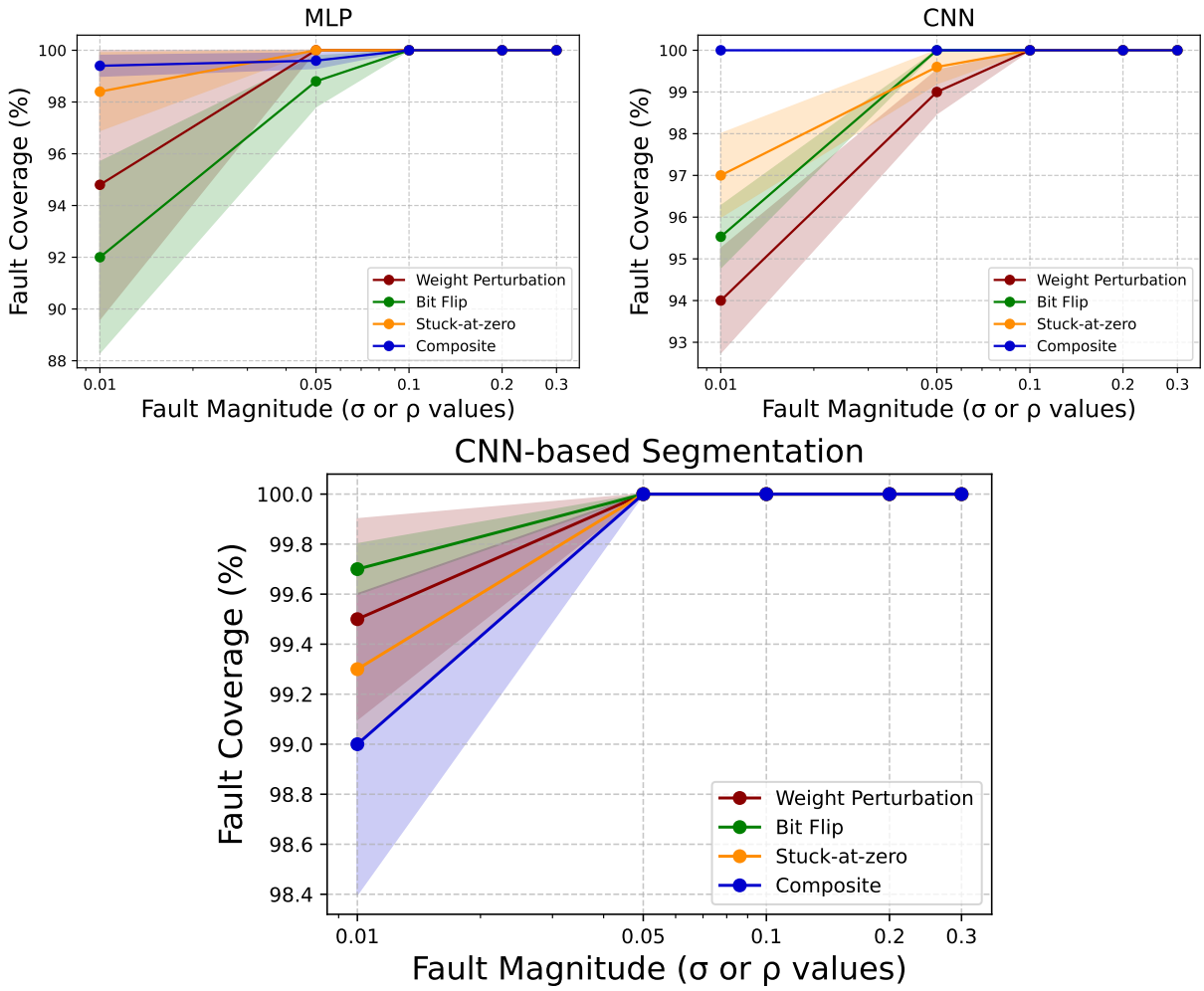

Figure 8.3.: Impact of Different Fault Injection on Fault Coverage Across Different Models.

As a function $d(\mathbf{x}, f)$ to measure the difference between the predicted classes, we use the squared norm difference between the softmax distributions, i.e., equation (8.1), exclusively for all our experiments. For optimization, we perform $T = 500$ update steps with the Adam [27] optimizer and estimate each gradient with $N = 500$ Monte Carlo samples. Computations were performed using 32-bit floating point weights (FP32 format) and implemented in *PyTorch* [49]. To find the learning rate, a grid search was performed on a log scale of 10^{-7} to 1. The MLP experiments were conducted on a machine equipped with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz and 16 GB of system RAM. The CNN experiments were performed using Google Colab Pro+ with access to an NVIDIA A100 Tensor Core GPU (40 GB HBM2) and up to approximately 100 GB of system RAM.

8.3.2. Experimental Results and Discussion

The effectiveness of the proposed test pattern generation approach is demonstrated through fault injection analysis for the MLP and CNN models across four different fault types: weight perturbation, bit flip, stuck-at, and composite faults (see Fig. 8.2).

The impact of these faults is assessed by analyzing inference accuracy degradation as the fault magnitude σ for weight perturbation and ρ for bit flip and stuck-at faults increases. For the MLP model, weight perturbation faults were introduced by adding Gaussian noise with standard deviation σ to p=5% of the weights.

Table 8.1.: Comparison of fault coverage across different test pattern generation approaches for various fault models, neural network architectures, and datasets. MLP experiments utilize 30 test patterns, while all CNN-based models use 40 test patterns. Each experiment is conducted using N=500 Monte Carlo samples and T=500 steps of Adam [27]. In the context of fault modeling, WP denotes weight perturbations, BF denotes bit flips, SA denotes stuck-at-zero faults, and CF refers to composite faults. Also, p is the perturbation percentage. For WP and CF: Gaussian noise with standard deviation σ is added to the weights. For BF and SA: ρ is the percent of flipped bit or stuck-at-zero neurons.

Experimental setup		**Test pattern approaches**		
Model and Dataset	**Fault model**	**Fault Coverage (%)**		
		Random	**Adv. [45]**	**Proposed**
MLP (MNIST)	WP(p=5%) σ=0.1	66	56	100
	BF(p=5%) ρ=0.1	79	76	100
	SA(p=5%) ρ=0.1	68	26	100
	CF(p=5%) σ=0.1 ρ=0.1	82	70	100
ConvNet-7 (CIFAR-10)	WP(p)=10% σ=0.05	89	54	100
	BF(p=10%) ρ=0.05	94	66	99.6
	SA(p=10%) ρ=0.05	80	60	99.8
	CF(p=10%) σ=0.05 ρ=0.05	66	77	100
VGG19 (CIFAR-10)	WP(p=15%) σ=0.1	44	36	100
	BF(p=15%) ρ=0.1	68	28	100
CNN-based Segmentation (CamVid)	WP(p=15%) σ=0.1	90	33	100
	BF(p=15%) ρ=0.1	88	46	100
	SA(p=15%) ρ=0.1	87	43	100
	CF(p=15%) σ=0.1 ρ=0.1	61	60	100

At low σ values, the accuracy already shows a decrease due to the smaller size of the model, even minor perturbations have a noticeable effect on the accuracy, and as σ increases, the degradation becomes more pronounced. At σ=0.2 and σ=0.3, the accuracy drops significantly, indicating that MLPs are highly sensitive to perturbations.

For bit flip and stuck-at faults, p=5% of the weights or neurons are first randomly selected for perturbation. For bit flip faults, each selected weight undergoes random bit-level changes, where a proportion ρ of its bits are flipped, i.e., ρ determines the percentage of bits within each selected weight

that are altered. In the case of stuck-at faults, each selected neuron has a probability ρ of being set to zero, simulating a neuron stuck-at a constant inactive state.

The bit flip results shows that the accuracy slowly decreases, but remains relatively high. This can be explained by the fact that bit flips might not significantly affect the model even when a bit flip changes the weight value. If the change is small, the neuron output might remain within the activation tolerance, so the network could still generalize well despite the small perturbation. Unlike bit flip faults, stuck-at faults, where a fraction of neurons were set to zero, cause a much sharper decline in accuracy. Here, a selected percentage of neurons is set to zero with probability ρ.

Compared to bit flips, stuck-at faults disrupt the model more severely, likely due to the complete loss of contribution from affected neurons, which appears to critically impair the network. Compared to weight perturbations, composite faults, which combine weight perturbations p=5%, bit flips (ρ=1%), and stuck-at faults (ρ=1%), initially show a gradual degradation in accuracy. This can be due to the fact that we only flip ρ=1% of the bit and only set ρ=1% of the neurons to zero. Increasing these values can severely disrupt the network, leading to rapid accuracy degradation. However, at higher σ values, the combined effect of multiple faults becomes more apparent, leading to severe accuracy degradation.

For the CNN model, the effect of weight perturbation faults is less severe compared to MLPs, demonstrating CNNs higher robustness to small perturbations. The accuracy remains relatively high at small values of σ and starts to decline beyond σ=0.1, with a sharp drop at σ=0.2. Compared to MLPs, CNNs have more parameters and hierarchical feature extraction, which may provide greater tolerance to minor parameter fluctuations.

For bit flips, p=10% of the weights were perturbed, leading to gradual accuracy degradation. However, at higher ρ values, CNN accuracy degraded rapidly, indicating that larger-scale bit flip faults significantly disrupt CNN performance. In terms of stuck-at faults (ρ=10% of neurons), CNN showed a stronger impact than in MLP, particularly at higher values of ρ, where accuracy drops sharply. This may be because CNNs rely on structured feature extraction and zeroing out certain activations leads to a loss of spatial information. Unlike MLPs, where weights can compensate for missing neurons, CNNs suffer from zeroed-out filters that remove important feature extraction capabilities. Consequently, when neurons are zeroed out, spatial information is lost, causing significant accuracy degradation. Composite faults combine p =10% weight perturbations with bit flips (ρ=1%), and stuck-at faults (ρ=1%) resulting in progressive accuracy loss, although at low perturbation levels, CNNs still maintained a degree of robustness.

For the CNN-based segmentation model, the impact of faults was more severe compared to the classification models. Accuracy degraded more rapidly, even for small perturbations, particularly for stuck-at and composite faults. This increased sensitivity arises because segmentation tasks depend on fine-grained spatial feature mappings, making them more vulnerable to neuron failures. Unlike classification tasks, where some redundant features can compensate for faults, segmentation relies on continuous spatial coherence, meaning even small perturbations can lead to significant misclassification in pixel-wise segmentation output.

Since the size of the set of test patterns (defining the available storage) is a user-defined parameter, Fig. 8.4 illustrates how increasing the number of test patterns impacts fault coverage. As expected, it can be seen that increasing the number of test patterns leads to a corresponding increase in fault coverage. Notably, under composite faults with p=5% perturbation using σ=0.01 for weight perturbations and ρ=0.01 for both stuck-at-zero and bit flip faults, full fault coverage was achieved with 30 test patterns for the MLP model. For the CNN model (ConvNet-7), more than 95% fault coverage was reached with 40 test patterns. This demonstrates the efficiency of the proposed approach, where even a small number of test patterns can produce a test set with high coverage. However, reducing the number of test patterns below these thresholds results in a noticeable drop in fault coverage, with the decline being steeper for CNNs compared to MLPs. This suggests that larger models, such as CNNs and CNN-based segmentation

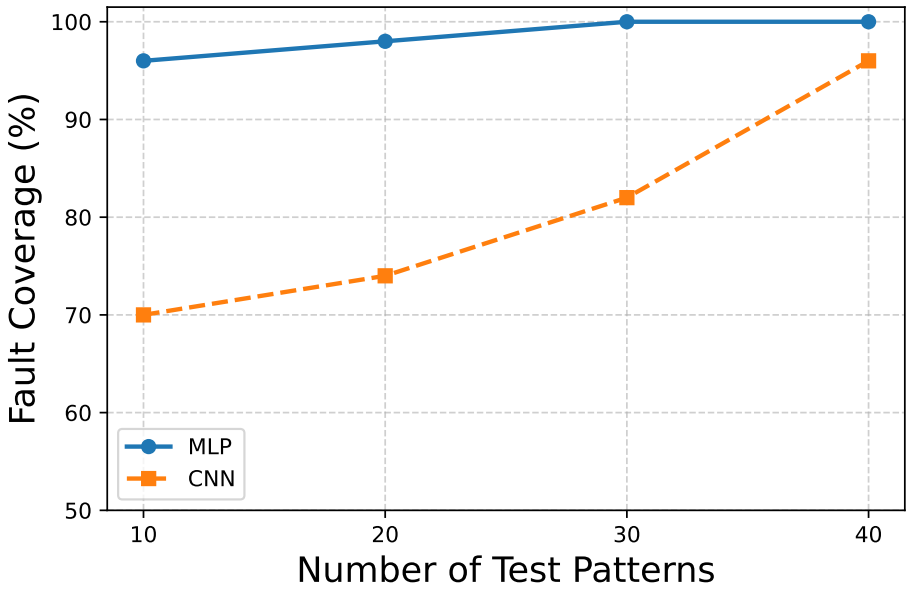

Figure 8.4.: Impact of test pattern size on fault coverage for MLP and CNN (ConvNet-7) models using Composite faults with p=5% perturbation using σ=0.01 for weight perturbations and ρ=0.01 for stuck-at-zero and bit flip faults.

networks, require a slightly larger test set to maintain high fault coverage, especially at low perturbation levels.

Furthermore, Fig. 8.3 shows the fault coverage analysis in the MLP, CNN, and CNN-based segmentation models, performed using 30 test patterns for MLP, 40 for CNN and segmentation, and N=500 Monte Carlo samples. The result demonstrated that the proposed test pattern effectively achieved high fault coverage across all models, even at low fault magnitudes. Specifically, small values of σ in weight perturbation and composite faults, as well as low values of ρ in bit flip and stuck-at faults, still resulted in high fault coverage. As the magnitude of the faults increased, the fault coverage increased, eventually reaching 100% for all fault types.

Finally, a comparison was made between random, adversarial, and our test pattern generation approach (see Table 8.1) to assess their effectiveness in achieving high fault coverage across different models, including MLP, CNN, and CNN-based segmentation. To further demonstrate the scalability of our method, we extended the experiments to include the VGG-19 model. The adversarial test patterns were generated using the FGSM, where small perturbations were introduced to the input images based on the gradient of the loss function with respect to the input. To ensure a fair and meaningful comparison, a set of random adversarial test patterns was generated, matching the number of test patterns used in our approach. The adversarial patterns were then applied as fault stimuli and the resulting fault coverage was measured.

The result showed that the proposed test pattern approach outperforms adversarial and random test patterns by consistently achieving higher fault coverage across all models, including MLP, CNN, CNN-based segmentation, and VGG-19. The method's superior performance across different models highlights its scalability and robustness in detecting faults.

8.4. Chapter Summary

In this chapter, a unified and fault-agnostic framework is presented for functional test pattern generation in DNNs, explicitly designed for in-field deployment scenarios. The proposed approach jointly optimizes a fixed-size set of test patterns under a probabilistic fault model, integrating storage and execution

constraints directly into the generation process. By embedding resource awareness into the optimization itself, the framework eliminates the need for post-generation compaction while maintaining high fault coverage across diverse fault types.

Extensive experimental evaluations across multiple network architectures and tasks, including image classification and semantic segmentation, demonstrate the scalability, robustness, and effectiveness of the proposed framework. In all scenarios considered, high fault coverage was achieved while respecting memory budgets.

9. Equivalence testing of model transformations.

9.1. Chapter Overview

In this chapter, the problem of functional equivalence checking is addressed for DNNs, with a focus on transformations introduced during deployment on resource-constrained hardware. As discussed earlier, DNNs are increasingly used in safety-critical domains such as industrial robotics, medical systems, and autonomous vehicles, where even minor deviations in behavior may lead to severe consequences [62].

Due to their high computational and memory demands [59], DNNs are often optimized for deployment using techniques such as quantization, pruning, and compression [32], [72]. Although training is typically performed with high precision, deployment relies on reduced precision or hardware-specific mappings, which can introduce subtle behavioral differences between the original and deployed models.

These deviations raise concerns for functional safety. It is therefore essential to verify whether a transformed or modified model preserves the intended behavior of the reference model. In particular, identifying concrete counterexample inputs that lead to different outputs provides valuable insight for debugging and validation [53], [57].

Traditional hardware verification and equivalence checking techniques are not directly applicable to DNNs due to their scale, non-linearity, and high-dimensional inputs [75]. Existing methods often rely on restrictive assumptions or detect differences in unrealistic regions of the input space [53], [57], [73]. In practice, it is more meaningful to search for discrepancies near real data samples.

To address this, a functional inequivalence detection approach is proposed. Instead of full formal verification, the method searches for counterexamples near realistic inputs by applying small perturbations to training samples. It aims to maximize differences in predicted class distributions while keeping perturbations semantically meaningful.

The approach is general and not limited to specific architectures or activation functions. The experimental results demonstrate its effectiveness in identifying behavioral differences between the original and transformed models, highlighting its relevance for validating deployment in safety-critical and edge AI systems.

The remainder of this chapter is structured as follows. Section 9.4 formalizes the problem and presents the method. Section 9.5 provides the experimental evaluation. Finally, the chapter concludes with a summary.

9.2. Neural Network Model Compression

Model compression aims to reduce runtime, memory consumption, and energy consumption, which is especially important for deployment on edge devices and hardware accelerators. Two widely used techniques are quantization and pruning.

Quantization reduces the numerical precision of the weights, biases, and activations. For example, converting parameters from 32-bit floating point to 8-bit integers reduces the model size by approximately a factor of four and can improve latency and energy efficiency. A common approach is post-training quantization, where the model is trained in floating point and quantized afterward. This approach is easy to apply but may introduce a small accuracy drop due to rounding and saturation effects.

Table 9.1.: Qualitative analysis between our proposed method and the state-of-the-art approaches.

Approaches	In (data) range counterexamples	Classification	Arbitrary activation functions
[74]	✗	✗	✗
[57]	✗	✗	✗
[73]	✗	✗	✗
[53]	✗	✓	✗
Proposed	✓	✓	✓

Pruning reduces the number of effective parameters while aiming to preserve the model accuracy. In weight pruning, individual weights are removed based on a criterion such as magnitude. A widely used strategy is magnitude-based pruning, motivated by ℓ_1 regularization (often referred to as ℓ_1 *weight decay*) [15], where weights with small absolute values are pruned after (or during) training.

In practice, quantization and pruning are often combined to achieve higher compression ratios while maintaining acceptable accuracy.

9.3. Related Work

Referring to Section 3.0.11, a qualitative comparison between existing approaches and the proposed method is summarized in Table 9.1.

Prior work relies primarily on piecewise linear assumptions, which restrict the applicability of these methods to networks with specific activation functions. As a result, modern architectures—such as transformer-based models that employ softmax and attention mechanisms [36]—are not supported by these approaches.

In addition, existing equivalence definitions are often not well aligned with classification tasks, where exact functional equivalence is less relevant than consistent predictive behavior. Another key limitation is that counterexamples identified by prior methods often lie in regions of the input space that are far from the underlying data distribution. This is problematic, as NNs are only reliably defined in regions supported by the training data.

Therefore, it is more meaningful to focus on detecting discrepancies in the vicinity of realistic data samples. The proposed approach addresses these limitations by avoiding restrictive architectural assumptions and by generating counterexamples that remain close to the data distribution, leading to a more practical and interpretable equivalence analysis.

9.4. Definitions of Equivalence for Neural Networks

Equivalence checking validates the correctness and consistency of NN models and their edge implementations. Therefore, identifying cases of non-equivalence is critical, as small differences can have catastrophic consequences in safety-critical applications. However, before developing a method for equivalence checking or counterexample generation, a proper definition of equivalence between two different NNs needs to be established. Previous works have discussed two potential definitions.

Definition 1: $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}'(\mathbf{x})$ are equivalent if

$$\|\mathrm{Net}(\mathbf{x}) - \mathrm{Net}'(\mathbf{x})\| \leq T \quad \text{for all inputs } \mathbf{x} \in \mathbb{X}.$$

In words, the outputs of the networks never deviate from each other by more than a threshold T for any admissible input $\mathbf{x} \in \mathbb{X}$.

Definition 2: $\text{Net}(\mathbf{x})$ and $\text{Net}'(\mathbf{x})$ are equivalent if

$$\underset{i}{\text{argmax}}\, \text{Net}(\mathbf{x})_i = \underset{i}{\text{argmax}}\, \text{Net}'(\mathbf{x})_i \quad \text{for all inputs } \mathbf{x} \in \mathbb{X}.$$

In words, both networks predict the same output class for all inputs $\mathbf{x} \in \mathbb{X}$. Previously proposed methods, such as [57], [73], mostly focus on *Definition 1* and propose methods to find counterexamples for this type of equivalence.

While *Definition 1* may be suitable for regression tasks, it might lack significance for classification. In case of classification, only the index of the output with the highest value determines the predicted class. The absolute value does not change the classification outcome. Hence, it is easy to construct cases in which *Definition 1*, when used for equivalence checking in a classification case, may be arbitrarily unrepresentative. For example, consider a case where $\text{Net}'(\mathbf{x}) := \text{Net}(\mathbf{x}) + C$, with $C \geq 2 \cdot T$ to the outputs of one network, i.e., the networks' outputs deviate by a fixed constant. In this case, *Definition 1* would indicate a non-equivalence for all $\mathbf{x} \in \mathbb{X}$. In contrast, the class label predicted by $\text{Net}'(\mathbf{x})$ and $\text{Net}(\mathbf{x})$ would remain unchanged and the networks are equivalent in the classification sense.

Similar examples could be constructed with other monotonic (order preserving) transformations. Additionally, depending on the networks, cases could be constructed where two networks would be equivalent according to *Definition 1*, while potentially predicting a different class label for all inputs of $\mathbf{x}$. For this, consider a two-class case, where $\text{Net}(\mathbf{x})$ predicts every example with a very low confidence (outputs close to $1/K$ for K classes). In this case, it might be possible to find a $\text{Net}'(\mathbf{x}) := 1 - \text{Net}(\mathbf{x})$ that predicts every label different to $\text{Net}(\mathbf{x})$, while still being equivalent according to *Definition 1*.

In this case, *Definition 2* should clearly be preferred, as it does not exhibit these deficiencies. Unfortunately, directly adapting *Definition 2* is likely too restrictive for any practical purpose. For example, even the slightest perturbation in the outputs of $\text{Net}(\mathbf{x})$ shifts the decision boundary (due to smoothness), and thus, examples $\mathbf{x}$ could be found for which *Definition 2* is violated. Hence, a more practical definition for NN equivalence in the classification setting is required.

In search of a suitable definition, we consider what is desired from the NN in the first place. Namely, to approximate given data $\mathcal{D}$. We write $\mathbf{x} \in \mathcal{D}$ if we only care about the $\mathbf{x}$ component in the tuple $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$. NN training is considered successful, if $\text{Net}(\mathbf{x}) \approx \mathbf{y}$ for all $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$. For any $\mathbf{x} \notin \mathcal{D}$, the correct label $\mathbf{y}$, and consequently the desired prediction, is not specified. Thus, strictly speaking and without considering further assumptions, different networks, as long as they agree for each data point tested, may be seen as indistinguishable.

However, to expect generalization, a certain smoothness in the behavior around the data points $\mathbf{x} \in \mathcal{D}$ is required. For example, to make sure to get similar outputs for slightly noisy versions of the same input $\mathbf{x}$. In consideration of this assumption, two NNs may be considered equivalent if they agree in a certain region of the input space in which the training data are located. Consequently, only points in these regions should be considered as candidates for counterexamples.

Based on these insights and to address the shortcomings of previous definitions, we propose a more comprehensive definition for classification tasks. The approach provides a practical and reliable framework for checking NN non-equivalence by generating meaningful counterexamples from relevant regions of the data space. It is also activation-function independent, allowing the method to work with a wide range of architectures, including advanced models like transformers that rely on non-linear activations for tasks such as calculating attention weights.

Definition 3: $\text{Net}(\mathbf{x})$ and $\text{Net}'(\mathbf{x})$ are equivalent if

$$\underset{i}{\text{argmax}}\, \text{Net}(\mathbf{p})_i = \underset{i}{\text{argmax}}\, \text{Net}'(\mathbf{p})_i \quad \text{for all } \mathbf{p} := \mathbf{x} + \eta$$
$$\text{with } \|\eta\| \leq T \text{ and } \mathbf{p} \in \mathbb{X} \text{ and } \mathbf{x} \in \mathcal{D}.$$

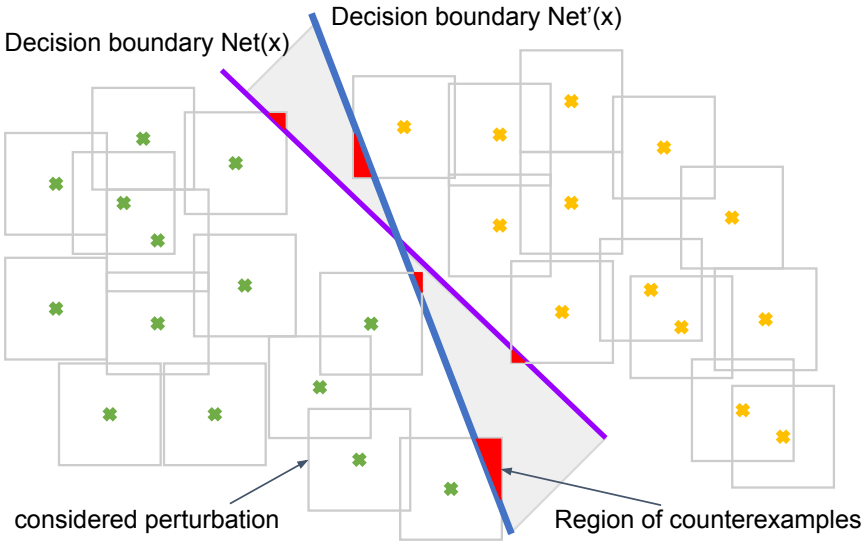

Figure 9.1.: An illustration of equivalence violations according to *Definition 3* (in red). The green and orange crosses denote data points of two different classes, while the purple and blue lines signify the decision boundaries of $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}'(\mathbf{x})$ respectively. Grey boxes around each data point (given by $\|\eta\|_\infty := \max_i\{\eta_i\}$ distances), indicate the considered perturbations. In the shaded region, the predictions of $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}'(\mathbf{x})$ do not agree. Thus, counterexamples may be found at the intersection (shown in red) of the boxes around the data examples and the shaded region.

In words, for limited, admissible perturbations $\mathbf{p}$ around data examples $\mathbf{x} \in \mathcal{D}$, both networks predict the same output class. The desired range of specification is thereby defined through T and the respective norm $\|\eta\|$. Higher values of T assume more generalization, and thus impose a stricter form of equivalence, as the region on which $\mathrm{Net}(\mathbf{x})$ and $\mathrm{Net}'(\mathbf{x})$ have to agree is larger. An illustration of the definition of equivalence can be seen in Figure 9.1. Blue and purple lines represent different decision boundaries of Net and Net′ which should be tested for equivalence according to *Definition 3*. The combined area of the grey boxes around the data points of one class represents the constraint region. For this region, we assume that the model is properly specified by the data, and equivalence should be checked. Note that for $T \to \infty$, *Definition 2* is recovered which would assume a violation of equivalence if any shaded region exists, or in other words, if the decision boundaries are not identical.

9.4.1. Counterexample Generation

Motivated by this definition of NN equivalence for classification, we propose an optimization problem to find counterexamples. In contrast to previous works, the objective is to find counterexamples close to the real data $\mathcal{D}$. This is motivated by the fact that outside of the data region, the behavior of the network is undefined/unspecified. Hence, given an input $\mathbf{x}$ from the data, we find counterexamples for the equivalence of two networks Net and Net′ by solving

$$\begin{aligned} &\underset{\eta}{\text{maximize}} \quad \left\| \mathrm{Softmax}\left(\mathrm{Net}(\mathbf{p})\right) - \mathrm{Softmax}\left(\mathrm{Net}'(\mathbf{p})\right) \right\|_2^2 \\ &\text{such that} \quad \mathbf{p} := \mathbf{x} + \eta \quad \text{and} \quad \|\eta\|_\infty \leq T \quad \text{for any} \quad \mathbf{x} \in \mathcal{D} \end{aligned} \tag{9.1}$$

The optimization objective is motivated by contrasting the outputs of two NNs, see [80]. Furthermore, through the additional constraints, we make sure that η, which denotes a limited perturbation from $\mathbf{x} \in \mathcal{D}$, identifies counterexamples in the range of the data. The size of the perturbation is given by $\|\eta\|_\infty$ and the parameter T denotes a threshold for this size and consequently the tolerance allowed.

As can be imagined from looking at Figure 9.1, the admissible region may be non-connected. The optimization problem is therefore difficult. Nevertheless, assuming that Net is differentiable by design (to employ backpropagation), the main objective is differentiable almost everywhere. Hence, iterative gradient-based optimization algorithms, such as (projected) gradient ascent, may be used to obtain a solution. To solve the optimization problem, several gradient-based searches should be started from different $\mathbf{x} \in \mathcal{D}$, or small, admissible perturbations of them. Additionally, promising starting points could be chosen based on their objective value. More specifically, if the objective function for the initial point displays a high value, the point is likely close to a region where the decision boundaries of Net and Net′ are further apart. Assuming smoothness, this may make a constraint violation in their vicinity more likely.

To respect the norm constraints, we project η on the locally admissible set (given by the norm constraint) after each update step. For the infinity norm, this can be done simply by clipping the absolute value of the entries element-wise. Similarly, if $\mathbb{X}$ also represents a box (e.g., pixel values in $[0, 255]$ for images), projections can be applied to ensure feasibility for the second constraint in the same way.

Algorithm 8 The counterexample generation algorithm for checking the equivalence of two networks Net and Net′. The threshold T, a given dataset $\mathcal{D}$, a step-size α, and the maximum number of steps K are assumed to be given. For simplicity, we describe only the update rule of gradient ascent, but also other iterative gradient-based optimizers could be used in line 13. The constraint $\mathbf{p} \in \mathbb{X}$ was omitted for clarity but can be addressed via an additional projection onto $\mathbb{X}$.

1: ### Function definitions
2: # Objective function
3: $f(\mathbf{p}) := \left\| \text{Softmax}\left(\text{Net}(\mathbf{p})\right) - \text{Softmax}\left(\text{Net}'(\mathbf{p})\right) \right\|_2^2$
4:
5: # Element-wise projection operation
6: $\text{Proj}\left[\eta\right] := \left[\text{Proj}\left[\eta\right]_0, \cdots, \text{Proj}\left[\eta\right]_i, \cdots\right]^\top$
7: $\text{Proj}\left[\eta\right]_i := \begin{cases} \eta_i & \text{if } |\eta_i| \leq T \\ \text{sign}(\eta_i) \cdot T & \text{otherwise} \end{cases}$
8:
9: ### Search for counterexamples
10: **for** $\mathbf{x}$ in $\mathcal{D}$ **do** ▷ for different starting positions
11: $\quad \eta^0 \leftarrow \mathbf{0}$
12: $\quad$ **for** k in $\{1, \cdots, K\}$ **do** ▷ steps per starting point
13: $\quad\quad \eta^{(k)} \leftarrow \text{Proj}\left[\eta^{(k-1)} + \alpha \cdot \nabla_\eta f\left(\mathbf{x} + \eta^{(k-1)}\right)\right]$
14: $\quad\quad \mathbf{p} \leftarrow \mathbf{x} + \eta^{(k)}$
15: $\quad\quad$ **if** $\underset{i}{\text{argmax}}\, \text{Net}(\mathbf{p})_i \neq \underset{i}{\text{argmax}}\, \text{Net}'(\mathbf{p})_i$ **then**
16: $\quad\quad\quad$ **return** $\mathbf{p}$ ▷ Counterexample found
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **end for**

If, at any point of the optimization, an admissible solution is found for which the Net and Net′ do not agree on the label, the procedure can be stopped, and the counterexample can be reported. The pseudo-code for finding a counterexample can be seen in Algorithm 8.

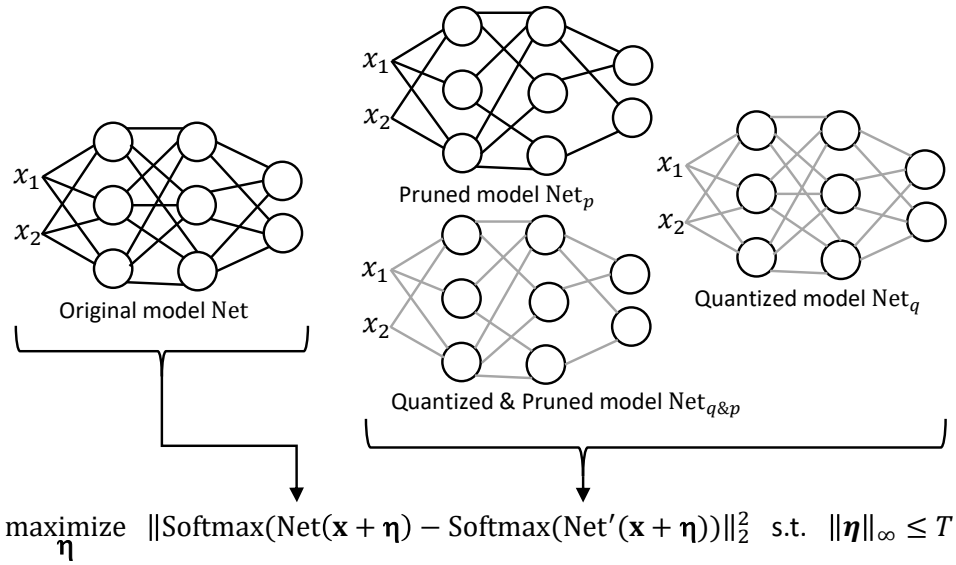

Figure 9.2.: Finding a counterexample through a limited perturbation η of an input $\mathbf{x} \in \mathcal{D}$ by maximizing the difference between the Softmax outputs of two networks Net and Net′. Here, compressed models: quantized (Net$_q$), pruned model (Net$_p$), or both (Net$_{p\&q}$) are tested for equivalence to the original network.

9.4.2. Practical Use Cases and Deployment Implications

There are several use case scenarios for such an equivalence checking method. Also, after identifying counterexamples and evidence of inequivalence between the original and compressed (edge) model, the next step is to determine the appropriate course of action, in order to repair the compressed edge model.

One potential use case scenario involves verifying the assertion made by a NN edge IP provider about an equivalence preserving transformation $\mathcal{T} : \text{Net} \mapsto \text{Net}'$ (mapping networks to networks), that should yield a network Net′, optimized for a certain edge hardware platform. Here, the equivalence checking approach can be used to assess whether the transformed network, $\text{Net}' = \mathcal{T}[\text{Net}]$, is equivalent to the original network Net according to *Definition 3*. If the methods employed in this verification process discover a counterexample, it indicates that the claim of equivalence preservation is false. In the event that a counterexample is indeed identified, it can be instrumental in the process of debugging and refining the transformation $\mathcal{T}$.

9.5. Experimental Evaluation and Analysis

9.5.1. Datasets and Experimental Setup

The experiments were conducted using datasets from MNIST [24], MedMNIST [87], and CIFAR-10 [25]. For MNIST, an MLP with four hidden layers of sizes $(512, 512, 256, 256)$ and an output layer of size 10 was employed, using sigmoid activations. For MedMNIST and CIFAR-10, a custom CNN architecture was used, consisting of two convolutional layers with 16 filters of size 3×3, followed by three convolutional layers with 64 filters, and three fully connected layers of size 512 with ReLU activations.

In addition, experiments were performed using the VGG-19 architecture [30]. The learning rate was determined through grid search over a logarithmic scale ranging from 10^{-7} to 1. All experiments were implemented in *PyTorch* [48], and optimization was carried out using the Adam optimizer [27]. The threshold T was set to $1/255$ (assuming the data were normalized to $[0, 1]$) for all experiments.

For the experiments, the 32-bit floating-point model was used, and quantization, pruning, as well as pruning combined with quantization were applied. Dynamic quantization was employed, whereby the

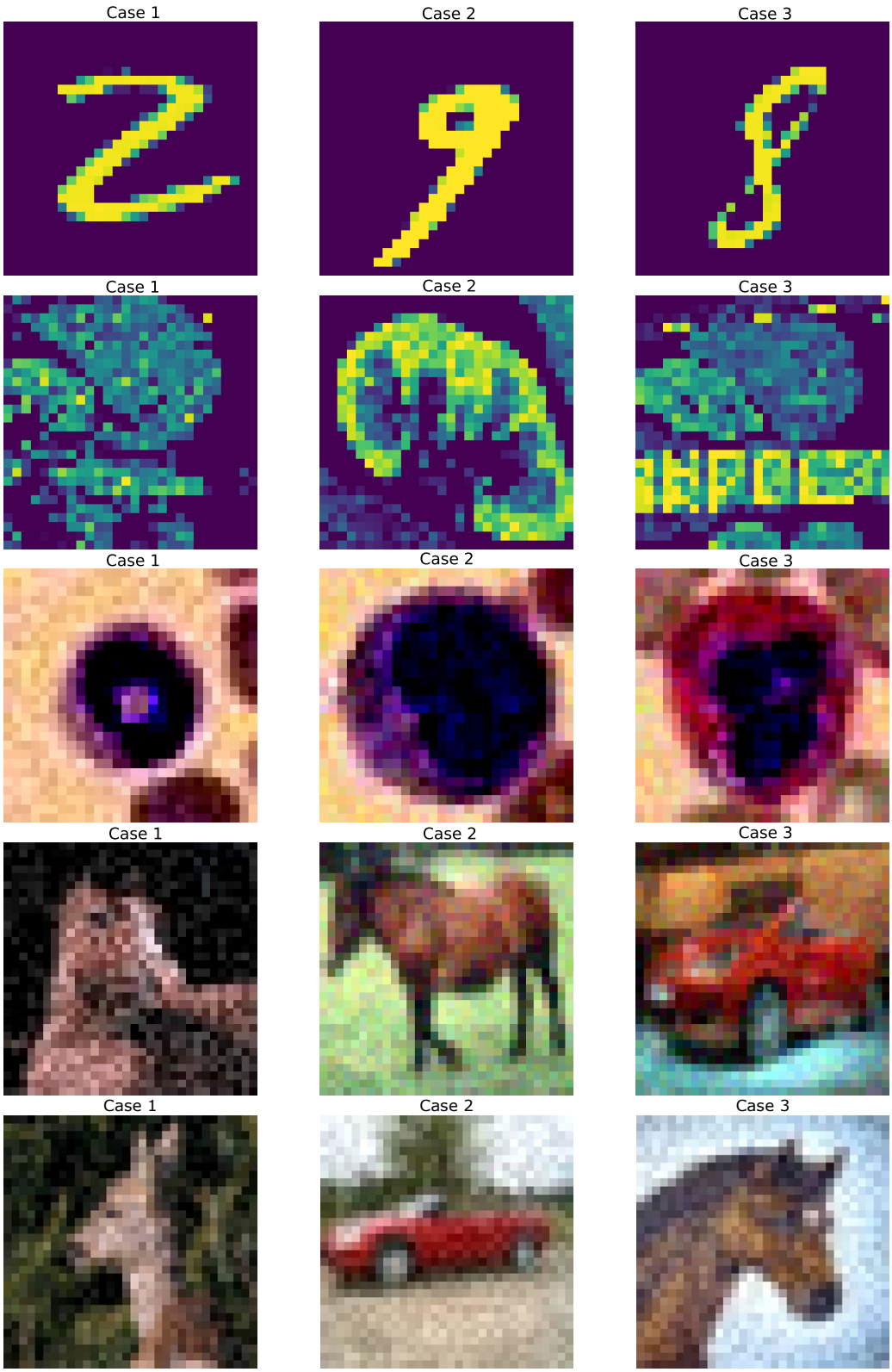

Figure 9.3.: Generated counterexamples for three cases scenarios (Original and Quantized models, Original and Pruned models, Original and Pruned + Quantized models) using in first row MLP (MNIST), in the second row: CNN (Medical OrganC MNIST), in the third row: CNN (Medical Blood MNIST) and in fourth row CNN (CIFAR-10) and in last row VGG19 (CIFAR-10)

Table 9.2.: The overall experimental results on different models and datasets. Q model refers to the Quantized model and Q label to its class label. P model refers to the Pruned model and P label to its class label. P+Q model represents the combined Pruned and Quantized model, with P+Q label as its class label. Model label refer to the original model label.

Experimental setup		Counter-examples generation														
Model and Dataset	Activation function	Case 1					Case 2					Case 3				
		Model	Q-model	Model label	Q label	Time (sec)	Model	P-model	Model label	P label	Time (Sec)	Model	P+Q model	Model label	P+Q label	Time (Sec)
		Train accuracy (%)					Train accuracy (%)					Train accuracy (%)				
MLP (MNIST)	Sigmoid	97.8	97.4	2	3	2606.4	96.8	96.8	9	7	536.9	97.9	99.3	3	5	1185.9
CNN (Medical OrganC MNIST)	ReLU	97.5	97.9	6	9	564.9	95.2	95.8	5	10	4.7	96.9	96.8	2	9	3.7
CNN (Blood MNIST)		95.6	96.1	7	2	506.8	95.4	95.9	2	7	8.3	95.2	96.0	1	7	106.0
CNN (CIFAR-10)		98.2	98.9	7	3	1566.0	96.4	96.1	7	2	10.6	96.5	96.8	1	9	7.8
VGG19 (CIFAR-10)		97.4	98.1	4	7	9883.0	98.2	98.4	1	9	10362.0	97.6	97.9	7	3	8963.0

weights of the trained NN model were converted from floating-point to INT8 precision during model execution. This conversion was performed using the dynamic quantization function, resulting in a quantized model. The dynamically quantized model was subsequently evaluated using the same training data.

For pruning, the L1 unstructured pruning method was applied, and the weights of the second, fourth, and sixth fully connected layers were pruned.

9.5.2. Experimental Results and Discussion

Referring to Section 9.4, the aim is to perform equivalence checking by identifying counterexamples. To locate such counterexamples (9.1) is solved according to Algorithm 8 using the Adam optimizer. The results in Table 9.2 present three scenarios. In the first scenario, the original model is compared with a quantized version. In the second scenario, it is compared with a compressed version. In the third scenario, the original model is compared with a combined pruned and quantized version.

As illustrated in Table 9.2, the training accuracy of both original and reduced models in all three scenarios is roughly the same. However, cases in which the original model and the compressed model disagree on the classification of any datapoint (as is indicated by a different accuracy), Algorithm 8 can already be stopped in the first iteration. Thus, to make the problem of finding counterexamples harder, all data points for which the models disagreed in their prediction were removed from $\mathcal{D}$. Additionally, the proposed method is applicable to networks composed of a broader set of functions, e.g., sigmod activations. This is in stark contrast to other state-of-the-art methods confined to only piece-wise linear functions.

The experiments show that numerous effective counterexamples are successfully identified, confirming the non-equivalence of the two models. Furthermore, counterexamples are effectively detected for the VGG-19 model, demonstrating the applicability of the approach to large-scale networks and highlighting its scalability. In Figure 9.3, one counterexample is presented for each of the three previously described scenarios across different model architectures and datasets. As intended, the generated examples remain close to real data and are easily recognizable by the human eye, suggesting that they lie within the data distribution where equivalent behavior would typically be expected. Table 9.2 reports the predicted labels of both models in each scenario, where it can be observed that different class labels are produced across various architectures and datasets, with the original model predicting the correct label while the compressed model produces incorrect predictions.

Additionally, as shown in Table 9.1, all considered aspects are covered by the proposed method, as it is applicable to networks composed of a broader range of functions, such as sigmoid activations. This stands in contrast to other methods that are restricted to piecewise-linear functions. Moreover, for equivalence checking, examples that closely resemble the actual dataset are identified. This is necessary because NNs are not well specified outside the data on which they are trained; therefore, only regions in the vicinity of the data points can be considered sufficiently specified.

Finally, the time needed to find the counterexample is reported. Evidently, finding counterexamples for larger, and more complicated models, requires more time as the computation of gradients is more costly. However, the approach remains significantly less time-consuming and more scalable compared to methods based on SAT or SMT solvers [34], [35].

9.6. Chapter Summary

In this chapter, the problem of model equivalence checking for NNs in classification settings was addressed. With the increasing deployment of optimized and compressed models on edge hardware platforms, verifying that such transformations preserve the intended functional behavior is of critical importance, particularly in safety-critical applications.

To this end, an adapted definition of NN equivalence was introduced, restricted to the relevant input region defined by the training data distribution. This data-aware notion of equivalence avoids impractical global guarantees and instead focuses on identifying meaningful behavioral deviations in regions where the model is expected to operate.

Building upon this definition, a counterexample generation method was proposed to identify inputs in the vicinity of real data points that induce divergent classification decisions between two models. The approach maximizes differences in the predicted class distributions while constraining perturbations to remain close to valid inputs, ensuring that the resulting counterexamples are both realistic and interpretable.

As a practical use case, equivalence between original (non-compressed) models and their compressed counterparts intended for edge deployment was evaluated. Experimental results across multiple architectures and datasets demonstrate that counterexamples can be effectively identified whenever inequivalence exists. This confirms that the proposed method provides a practical tool for validating model transformations, debugging deployment pipelines, and ensuring functional consistency in safety-critical DNNs.

10. Conclusion and Perspectives

10.1. Conclusion

This thesis addressed the problem of functional reliability and in-field testing of DNNs deployed on specialized hardware accelerators, with a particular focus on storage efficiency, scalability, and practical deployability under real-world constraints.

As DNNs are increasingly used in safety-critical and resource-constrained environments, traditional structural testing techniques alone are not sufficient. Functional testing becomes essential to ensure correct behavior under hardware faults, model transformations, compression, and quantization. However, applying functional testing to DNNs introduces fundamental challenges: the absence of explicit functional specifications, enormous input dimensionality, combinatorial fault scenarios, and stringent storage and runtime constraints in edge environments.

These challenges are addressed in this thesis through a sequence of progressively developed methodologies for real-world deployment.

First, an ATPGs framework for DNNs was introduced. Structural faults such as neuron- and filter-level SAFs were evaluated by their functional impact on inference behavior. By maximizing output deviations in the softmax distribution, test patterns were generated that induce misclassification in the presence of faults. Heuristic and cluster-based compaction strategies were proposed to reduce testing overhead while maintaining full fault coverage.

Second, the framework was extended to multiple fault models. Since realistic hardware accelerators are susceptible to multiple fault scenarios, a joint optimization strategy was developed to generate compact test patterns that cover multiple fault instances simultaneously. This significantly improves scalability and reduces the number of required test patterns.

Third, one of the central bottlenecks of in-field testing—test storage cost—was addressed through two different approaches:

- **Neural Built-In Self-Test**, which employs a lightweight generator network to produce high-coverage test patterns on demand from a seeded random source, dramatically reducing memory requirements.
- **Basis-Vector-based compressed test generation**, where test patterns are represented as linear combinations of basis vectors, allowing reconstruction from compact coefficient representations and achieving substantial compression ratios.

Both approaches demonstrated that storage efficiency can be integrated directly into the test generation process rather than treated as a post-processing step.

Fourth, an evolutionary optimization framework based on CMA-ES was proposed, which allows joint optimization of the fault coverage and test-set compactness of the test set in a single loop. By addressing the non-differentiable nature of test-set size, this method directly generated compact suites without separate compaction, achieving high coverage with significantly fewer patterns.

These developments were unified into a fault-agnostic storage framework for in-field functional testing, where test generation is formulated under explicit storage and runtime constraints and evaluated under probabilistic fault models.

Finally, the thesis extended beyond fault detection to model equivalence checking, addressing the need to verify that compressed, quantized, or hardware-mapped models remain behaviorally consistent with

their original versions. A data-aware notion of equivalence was introduced along with a counterexample search strategy capable of identifying meaningful divergences near realistic data samples.

Across all chapters, experimental validation on multiple architectures, datasets, and fault types demonstrated:

- High fault coverage (often reaching 100% coverage),
- Strong generalization to unseen faults,
- Significant reductions in test set size,
- Storage reductions extending up to 300×,
- Practical suitability for in-field and edge deployment.

Together, this thesis establishes a comprehensive and scalable framework for the validation of the functional reliability of hardware systems based on DNN, bridging optimization techniques, compressed representations, and deployment constraints.

10.2. Future Perspective

Although this work establishes a foundation for in-field functional testing of DNN accelerators, several promising research directions remain open.

10.2.1. Hardware-Software Co-Design for Testing

This work highlights the importance of considering both algorithmic and hardware aspects when designing test methodologies. In particular, the concept of NBISTs suggests a tighter integration between test generation and hardware implementation.

Future work may explore:

- Co-design of test pattern generators that explicitly account for hardware characteristics such as MAC reuse, memory hierarchy, and dataflow,
- Lightweight architectural support for in-field self-test and improved observability,

with the goal of enabling efficient and scalable self-testing tailored to specific accelerator architectures.

This direction is motivated by the observation that, in hardware implementations, faults are tied to physical resources and can be amplified through reuse, making hardware-aware testing particularly effective.

10.2.2. Extending to Transformer-Based Models

Extending the proposed methodology to modern architectures such as transformer-based models introduces additional challenges due to their discrete and sequential nature. Unlike conventional NNs operating on continuous input spaces, transformer models process sequences of tokens, resulting in a combinatorial and non-differentiable search space.

To address this, two potential directions can be considered:

- **Embedding-Level Test Token Optimization:** Introduce dedicated test tokens with learnable embeddings that are optimized to expose faults. This enables the application of continuous optimization techniques, at the cost of minor architectural modifications.

- **Continuous Relaxation of Token Sequences:** Perform optimization directly in the embedding space by generating sequences of continuous vectors, followed by a projection onto valid token embeddings. This avoids architectural changes but leads to a more challenging optimization problem.

Both approaches aim to bridge the gap between discrete token representations and continuous optimization methods, enabling effective test generation for transformer-based models.

Final Perspective

As AIs systems are increasingly deployed in safety-critical applications, ensuring their reliability under hardware imperfections and deployment transformations becomes essential.

This thesis demonstrates that:

- Functional testing for DNN accelerators can be made scalable,
- Storage constraints can be incorporated into test generation,
- High fault coverage can be achieved with compact test representations,
- Model transformations can be analyzed through data-driven equivalence checking,

Overall, the presented methodologies contribute toward improving the reliability and practical deployment of DNN-based systems on real hardware platforms.

List of own publications included in this thesis

[80] D. A. Moussa, M. Hefenbrock, C. Münch, and M. Tahoori, "Automatic test pattern generation and compaction for deep neural networks", in *Asia and South Pacific Design Automation Conference (ASPDAC)*, IEEE, 2023, pp. 436–441.

[81] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Compact test pattern generation for multiple faults in deep neural networks", in *Design, Automation & Test in Europe (DATE)*, IEEE, 2023, pp. 1–2.

[90] 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.

[93] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Compressed test pattern generation for deep neural networks", *IEEE Transactions on Computers*, vol. 74, no. 1, pp. 307–315, 2025.

[94] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Detecting non-equivalence in neural networks through in-distribution counterexample generation", *IEEE Embedded Systems Letters*, 2025, to appear.

[95] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Functional test generation for in-field testing of deep learning models with test storage constraints", in *IEEE International Test Conference (ITC)*, to appear, IEEE, 2025.

[96] T. Gheshlaghi, D. A. Moussa, M. Hefenbrock, and M. B. Tahoori, "Functional test pattern generation and compaction for dnns using evolutionary optimization", in *Proceedings of the IEEE VLSI Test Symposium (VTS)*, To appear, IEEE, 2026.

[98] D. A. Moussa, M. Hefenbrock, and M. B. Tahoori, "Neural built-in self-test for deep neural networks", in *Proceedings of the IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Under submission, IEEE, 2026.

List of other publications not included in this thesis

[97] T. Gheshlaghi, A. Studt, P. Pal, D. A. Moussa, M. Hefenbrock, M. Beigl, and M. B. Tahoori, "Diagnostic test generation for fault localization in printed neuromorphic circuits", in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Short paper, IEEE, 2026.

Part III.

Appendix

Bibliography

[1] H. Robbins and S. Monro, "A stochastic approximation method", *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[2] H. T. Kung and C. E. Leiserson, "Systolic arrays for (vlsi)", *Sparse Matrix Proceedings*, 1978.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0.

[4] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis", *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987.

[5] V. D. Agarwal and E. Cerny, "Store-and-generate built-in testing approach", *IEEE Design & Test*, 1989.

[6] J.-H. Chang et al., "Test set compaction using genetic algorithms", in *IEEE International Test Conference (ITC)*, 1995.

[7] N. A. Touba and E. J. McCluskey, "Bit-fixing techniques for deterministic bist", *IEEE Transactions on CAD*, 1996.

[8] H.-J. Wunderlich and G. Kiefer, "Bit-flipping bist", in *IEEE International Test Conference (ITC)*, 1996, pp. 337–346.

[9] F. Somenzi, "Binary decision diagrams", *NATO ASI Series F: Computer and Systems Sciences*, vol. 173, pp. 303–368, 1999.

[10] D. MacMillen, "An industrial view of equivalence checking", *IEEE Design & Test*, 2000.

[11] P. C. Maxwell et al., "Comparing functional and structural test quality", *IEEE International Test Conference (ITC)*, 2000.

[12] I. Pomeranz and S. M. Reddy, "Static test compaction for combinational circuits", *IEEE Transactions on Computers*, 2001.

[13] N. Hansen and A. Ostermeier, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)", *Evolutionary Computation*, vol. 11, no. 1, pp. 1–18, 2003.

[14] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2003.

[15] A. Y. Ng, "Feature selection, l1 vs. l2 regularization, and rotational invariance", in *Proceedings of the Twenty-First International Conference on Machine Learning (ICML)*, ACM, 2004, p. 78.

[16] A. Veneris and A. El-Maleh, "Functional fault modeling for logic circuits", in *IEEE International Test Conference (ITC)*, 2005, pp. 1–10.

[17] J. Marques-Silva, "Practical applications of sat solvers", *IEEE Design & Test*, 2008.

[18] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database", in *Proceedings of the British Machine Vision Conference (BMVC)*, BMVA Press, 2009, pp. 1–11.

[19] I. Pomeranz and S. M. Reddy, "Forward-looking reverse order fault simulation for test compaction", *IEEE Transactions on CAD*, 2009.

[20] A. Krizhevsky, G. Hinton, et al., "Convolutional deep belief networks on cifar-10", *Unpublished manuscript*, vol. 40, no. 7, pp. 1–9, 2010.

[21] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A. Y. Ng, et al., "Reading digits in natural images with unsupervised feature learning", in *NIPS workshop on deep learning and unsupervised feature learning*, Granada, vol. 2011, 2011, p. 7.

[22] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[23] S. Biswas and B. Cory, "An industrial study of system-level test", *IEEE Design & Test of Computers*, vol. 29, no. 1, pp. 19–27, 2012.

[24] L. Deng, "The mnist database of handwritten digit images", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[25] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset", *Online*, 2014.

[26] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples", in *International Conference on Learning Representations (ICLR)*, 2015.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *CoRR*, vol. abs/1412.6980, 2015.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", in *International Conference on Learning Representations (ICLR)*, 2015.

[29] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge", *CS 231N*, vol. 7, no. 7, p. 3, 2015.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *International Conference on Learning Representations (ICLR)*, 2015, arXiv:1409.1556.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[32] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks", *arXiv preprint arXiv:1710.09282*, 2017.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, et al., "In-datacenter performance analysis of a tensor processing unit", in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12. DOI: 10.1145/3079856.3080246.

[34] R. Majumdar and V. Kuncak, *Computer-Aided Verification*. Springer, 2017.

[35] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming", *arXiv preprint arXiv:1711.07356*, 2017.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need", in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, Curran Associates, Inc., 2017.

[37] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization", *International Conference on Learning Representations (ICLR)*, 2017.

[38] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2881–2890.

[39] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 633–641.

[40] C. Li, M. Hu, Y. Li, H. Jiang, et al., "Analogue signal and image processing with large memristor crossbars", *Nature Electronics*, vol. 1, pp. 52–59, 2018.

[41] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training", *arXiv preprint arXiv:1807.05118*, 2018.

[42] S. Markidis, S. W. Der Chien, E. Laure, et al., "Nvidia tensor core programmability, performance and precision", *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018.

[43] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework", in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2019, pp. 2623–2631. DOI: 10.1145/3292500.3330701.

[44] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network", in *IEEE Latin American Test Symposium (LATS)*, 2019.

[45] W. Li, Y. Wang, H. Li, and X. Li, "Rramedy: Protecting reram-based neural network from permanent and soft faults during its lifetime", in *IEEE International Conference on Computer Design (ICCD)*, 2019, pp. 91–99.

[46] B. Luo, Y. Li, L. Wei, and Q. Xu, "On functional test generation for deep neural network ips", in *Design, Automation & Test in Europe (DATE)*, 2019, pp. 1010–1015.

[47] Y. Luo, Y. Zhang, et al., "On functional test generation for deep neural networks", in *Design, Automation and Test in Europe (DATE)*, 2019, pp. 1422–1427.

[48] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, et al., "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[50] A. Bosio, G. Di Natale, et al., "Reliability analysis of deep neural network architectures under permanent faults", *IEEE Transactions on Emerging Topics in Computing*, 2020.

[51] A. Chaudhuri, J. Talukdar, F. Su, and K. Chakrabarty, "Functional criticality classification of structural faults in ai accelerators", in *IEEE International Test Conference (ITC)*, 2020, pp. 1–5.

[52] T.-Y. Hsieh et al., "Fault modeling and testing of spiking neural networks", in *IEEE International Test Conference (ITC)*, 2020.

[53] M. Kleine Büning, P. Kern, and C. Sinz, "Verifying equivalence properties of neural networks with relu activation functions", in *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, Springer, 2020, pp. 868–884.

[54] Q. Liu et al., "Cross-entropy based fault detection for deep neural networks", in *IEEE International Test Conference (ITC)*, 2020.

[55] Q. Liu et al., "Monitoring and testing deep neural networks for fault detection", *IEEE Transactions on Computers*, 2020.

[56] Q. Liu, T. Liu, Z. Liu, W. Wen, and C. Yang, "Monitoring the health of emerging neural network accelerators with cost-effective concurrent test", in *ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[57] B. Paulsen, J. Wang, J. Wang, and C. Wang, "Neurodiff: Scalable differential verification of neural networks using fine-grained approximation", in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2020, pp. 784–796.

[58] P. Wu, X. Sun, Z. Zhao, H. Wang, S. Pan, and B. Schuller, "Classification of lung nodules based on deep residual networks and migration learning", *Computational Intelligence and Neuroscience*, vol. 2020, pp. 1–10, 2020.

[59] J. Zhang and J. Li, "Testing and verification of neural-network-based safety-critical control software: A systematic literature review", *Information and Software Technology*, vol. 123, p. 106 296, 2020.

[60] D. Appello, H. H. Chen, M. Sauer, I. Polian, P. Bernardi, and M. S. Reorda, "System-level test: State of the art and challenges", in *Proceedings of the IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2021, pp. 1–7.

[61] A. Chaudhuri, J. Talukdar, J. Jung, G.-J. Nam, and K. Chakrabarty, "Fault-criticality assessment for ai accelerators using graph convolutional networks", in *Design, Automation & Test in Europe (DATE)*, 2021, pp. 1596–1599.

[62] D. Corsi, E. Marchesini, and A. Farinelli, "Formal verification of neural networks for safety-critical tasks in deep reinforcement learning", in *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, PMLR, 2021, pp. 333–343.

[63] S. Eggersglüß, S. Milewski, J. Rajski, and J. Tyszer, "On reduction of deterministic test pattern sets", in *IEEE International Test Conference (ITC)*, 2021, pp. 260–267.

[64] G. Hantos, D. Flynn, and M. P. Y. Desmulliez, "Built-in self-test (bist) methods for mems: A review", *Micromachines*, vol. 12, no. 1, 2021, ISSN: 2072-666X. DOI: 10.3390/mi12010040. [Online]. Available: https://www.mdpi.com/2072-666X/12/1/40.

[65] K. He, X. Chen, S. Xie, Y. Li, P. Doll'ar, and R. Girshick, "Masked autoencoders are scalable vision learners", *arXiv:2111.06377*, 2021.

[66] B. Hsu et al., "Accelerating ai at the edge with the google edge tpu", in *IEEE Hot Chips Symposium*, 2021.

[67] J. Lee et al., "A scalable deep learning accelerator architecture for mobile npus", *IEEE Transactions on Circuits and Systems*, 2021.

[68] F. Meng, F. Hosseini, and C. Yang, "A self-test framework for detecting fault-induced accuracy drop in neural network accelerators", in *ASPDAC*, 2021, pp. 722–727.

[69] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions", *SN Computer Science*, vol. 2, no. 6, pp. 1–20, 2021.

[70] N. Sharma, R. Sharma, and N. Jindal, "Machine learning and deep learning applications – a vision", *Global Transitions Proceedings*, vol. 2, no. 1, pp. 24–28, 2021, ISSN: 2666-285X.

[71] P. Stephan and H.-J. Wunderlich, "Reduction of deterministic test pattern sets for combinational circuits", *IEEE Transactions on CAD*, 2021.

[72] B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, "Enabling machine learning on the edge using sram-conserving efficient neural networks execution approach", in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, Springer, 2021, pp. 20–35.

[73] S. Teuber, M. Kleine Büning, P. Kern, and C. Sinz, "Geometric path enumeration for equivalence verification of neural networks", in *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, 2021, pp. 200–208.

[74] P. Kern, M. Kleine Büning, and C. Sinz, "Optimized symbolic interval propagation for neural network verification", *arXiv preprint arXiv:2212.08567*, 2022.

[75] I. Khmelnitsky, D. Neider, R. Roy, X. Xie, B. Barbot, B. Bollig, A. Finkel, S. Haddad, M. Leucker, and L. Ye, "Analysis of recurrent neural networks via property-directed verification of surrogate models", *International Journal on Software Tools for Technology Transfer*, 2022.

[76] S. Kundu and K. Basu, "Detecting functional safety violations in online ai accelerators", in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2022, pp. 1–4.

[77] Z. Aghababaeyan, M. Abdellatif, L. Briand, R. S, and M. Bagherzadeh, "Black-box testing of deep neural networks through test case diversity", *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3182–3204, 2023.

[78] A. Chaudhuri et al., "Functional testing of ai accelerators using bayesian optimization", in *IEEE International Test Conference (ITC)*, 2023.

[79] A. Chaudhuri, C.-Y. Chen, and K. Chakrabarty, *Recent Advances in Testing Techniques for AI Hardware Accelerators*. Jan. 2023, ISBN: 978-1-63828-240-2. DOI: 10.1561/9781638282419.

[80] D. A. Moussa, M. Hefenbrock, C. Münch, and M. Tahoori, "Automatic test pattern generation and compaction for deep neural networks", in *Asia and South Pacific Design Automation Conference (ASPDAC)*, IEEE, 2023, pp. 436–441.

[81] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Compact test pattern generation for multiple faults in deep neural networks", in *Design, Automation & Test in Europe (DATE)*, IEEE, 2023, pp. 1–2.

[82] A. Ruospo, G. Gavarini, A. Porsia, M. S. Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale, "Image test libraries for the on-line self-test of functional units in gpus running cnns", in *2023 IEEE European Test Symposium (ETS)*, IEEE, 2023, pp. 1–6.

[83] U. Solangi, A. Tunio, M. Hussain, A. Memon, A. Ayaz, and G. Hussain, "A review of structural testing methods for asic-based ai accelerators", *International Journal of Computer Science and Network Security*, vol. 23, no. 1, p. 103, 2023.

[84] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of ai hardware: Survey, trends, challenges, and perspectives", *IEEE Design & Test*, vol. 40, no. 2, pp. 8–58, 2023.

[85] N. E. Toklu et al., "Evotorch: Scalable evolutionary computation in pytorch", *arXiv preprint arXiv:2305.XXXX*, 2023.

[86] M. Turco et al., "Uncovering hard-to-detect permanent faults in convolutional networks via evolutionary search", *IEEE Transactions on Computers*, 2023.

[87] J. Yang, R. Shi, D. Wei, Z. Liu, L. Zhao, B. Ke, H. Pfister, and B. Ni, "Medmnist v2: A large-scale lightweight benchmark for 2d and 3d biomedical image classification", *Scientific Data*, vol. 10, no. 1, p. 41, 2023.

[88] B. B. Bhattacharya, D. K. Das, S. Chatterjee, and H. Rahaman, "Fault testing in ai-accelerators: A review", in *2024 IEEE 33rd Asian Test Symposium (ATS)*, 2024, pp. 1–6. DOI: 10.1109/ATS64447.2024.10915272.

[89] X. Fu et al., "Distributed training and inference on aws trainium", *IEEE Micro*, 2024.

[90] 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.

[91] L. Chen, X. Song, A. Song, B. Chen, J. Lv, and Y. Sun, "Fas: Fast ann-snn conversion for spiking large language models", *arXiv preprint arXiv:2502.04405*, 2025.

[92] T. Gheshlaghi, P. Pal, A. Studt, M. Hefenbrock, M. Beigl, and M. B. Tahoori, "Automatic test pattern generation for printed neuromorphic circuits", in *Proceedings of the 30th IEEE European Test Symposium (ETS)*, to appear, Tallinn, Estonia: IEEE, 2025.

[93] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Compressed test pattern generation for deep neural networks", *IEEE Transactions on Computers*, vol. 74, no. 1, pp. 307–315, 2025.

[94] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Detecting non-equivalence in neural networks through in-distribution counterexample generation", *IEEE Embedded Systems Letters*, 2025, to appear.

[95] D. A. Moussa, M. Hefenbrock, and M. Tahoori, "Functional test generation for in-field testing of deep learning models with test storage constraints", in *IEEE International Test Conference (ITC)*, to appear, IEEE, 2025.

[96] T. Gheshlaghi, D. A. Moussa, M. Hefenbrock, and M. B. Tahoori, "Functional test pattern generation and compaction for dnns using evolutionary optimization", in *Proceedings of the IEEE VLSI Test Symposium (VTS)*, To appear, IEEE, 2026.

[97] T. Gheshlaghi, A. Studt, P. Pal, D. A. Moussa, M. Hefenbrock, M. Beigl, and M. B. Tahoori, "Diagnostic test generation for fault localization in printed neuromorphic circuits", in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Short paper, IEEE, 2026.

[98] D. A. Moussa, M. Hefenbrock, and M. B. Tahoori, "Neural built-in self-test for deep neural networks", in *Proceedings of the IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Under submission, IEEE, 2026.

[99] Y. Chen, *Pytorch cifar models*, `https://github.com/chenyaofo/pytorch-cifar-models`, Accessed: 2025-5-17.

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence.

AI-HA AI Hardware Accelerator.

ASIC Application-Specific Integrated Circuit.

ATPG Automatic Test Pattern Generation.

BDD Binary Decision Diagram.

BIST Built-In Self-Test.

cGAN Conditional Generative Adversarial Network.

CMA-ES Covariance Matrix Adaptation Evolution Strategy.

CNN Convolutional Neural Network.

DFT Design-For-Test.

DNN Deep Neural Network.

EA Evolutionary Algorithms.

FGSM Fast Gradient Sign Method.

GAN Generative Adversarial Network.

GPU Graphics Processing Unit.

IP Intellectual Property.

IST In-System Test.

ITL Image Test Library.

MAC Multiply-Accumulate.

MLP multilayer perceptron.

NBIST Neural Built-In Self-Test.

NN Neural Network.

NUT Network Under Test.

ReLU Rectified Linear Unit.

SAF Stuck-at faults.

SAT Boolean Satisfiability.

SGD Stochastic Gradient Descent.

SiLU Sigmoid Linear Unit.

SNN Spiking Neural Network.

SoC system on a chip.

TPU Tensor Processing Unit.

VGG Visual Geometry Group.