TestifAI: A Comprehensive Testing Framework For Safe AI

By

Arooj Arif

aa3506phd

Mini-thesis is submitted for the probation review of PhD

July 2024

Northeastern University London, London - UK

Spring, 2024

Final Approval

This Mini-thesis titled

TestifAI: A Comprehensive Testing Framework For Safe AI

By

Arooj Arif

aa3506phd

has been approved

For the Northeastern University, London

Chair:____

Alex Freitas,
Professor, Department of Computer Science,
University of Kent, Kent

Supervisor:____

Alexandros Koliousis,
Associate Professor, Department of Computer Science,
Northeastern University London, UK

Co-Supervisor:____

Elena Botoeva,
Lecturer, Department of Computer Science,
University of Kent, Kent

ABSTRACT

Deep neural networks (DNNs) models are critical in high-stake domains such as autonomous driving, medical diagnostics, and security systems, where their deployment in real-world scenarios requires rigorous robustness testing due to diverse environmental conditions. Traditional metrics like neuron coverage, while essential, do not fully capture all corner cases, which can lead to unexpected model failures. To address this gap, this research introduces a comprehensive testing framework that enhances the correctness evaluation of models through a structured five-stage process. The first stage is specification, defines essential system properties to guide the entire testing process and ensure comprehensive coverage. The second sampling stage, gathering relevant samples for exhaustive model testing. In the test case generation stage, the defined properties are applied to create targeted test scenarios. The testing and probabilistic graph stage validates the effectiveness of these test cases and conducts robustness assessments both locally (within individual category) and globally (across multiple scenarios), employing a Problog for detailed probabilistic and quantitative analysis of performance. The final stage is error summarisation, compiles and analyzes recorded errors to generate actionable graphical error reports and recommendations, thus guiding the refinement of models. This framework not only fills existing gaps in DNNs testing but also supports the development of models that are correct across varied environmental conditions.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview

Deep Neural Networks (DNNs) are increasingly being used in diverse applications due to their ability to match or exceed human-level performance. The availability of large datasets, fast computing methods, and their high performance has paved the way for DNNs in safety-critical applications such as autonomous driving, medical diagnosis, and security. The safety-critical nature of such applications makes it imperative to adequately test these DNNs before deployment. However, unlike traditional software, DNNs do not have a clear control-flow structure. They learn their decision policy through training on large datasets, adjusting parameters gradually using various methods to achieve the desired accuracy. Consequently, traditional software testing methods like functional coverage and branch coverage cannot be applied to DNNs, thus challenging their use in safety-critical applications.

Recent work, discussed in Chapter II, has focused on developing testing frameworks for DNNs. These methods suffer from certain limitations, as discussed in the challenges section. In our work, we aim to overcome these limitations and build a fast, scalable, efficient, and generalizable testing framework for deep neural networks.

In this section of the thesis, the background and motivation, research questions, contributions, and organization of the thesis are presented.

1.1.1 Background and Motivation

In recent years, DNNs have made remarkable progress in achieving human-level performance. With the broader deployment of DNNs in various safety-critical systems like autonomous vehicles, healthcare, and avionics, concerns over their safety and trustworthiness have been raised, particularly highlighted by incidents involving self-driving cars.

An important requirement for DNNs is robustness against input perturbations. DNNs have been shown to lack robustness due to their susceptibility to adversarial examples, where small modifications to an input, sometimes imperceptible to humans, can make the network unstable.

This thesis examines existing testing methods for DNNs, opportunities for improvement, and the need for a fast, scalable, generalizable end-to-end testing method.

Coverage criteria for traditional software programs, such as code coverage and branch coverage, ensure that all parts of the logic in the program have been tested by at least one test input and all conditions have been tested to independently affect the entailing decisions. Similarly, any coverage criterion for DNNs must ensure that all parts of the internal decision-making structure of the DNN have been exercised by at least one test input.

Generating or selecting test inputs in a guided manner usually has two major goals: maximizing the number of uncovered faults and maximizing the coverage.

Testing DNNs for correctness involves verifying behaviors against a ground truth or oracle. The traditional approach of collecting and manually labeling real-world data is labor-intensive. Another method compares outputs across multiple DNNs for the same task, identifying discrepancies as corner cases. However, this can misclassify inputs if all models agree, due to shared biases or errors. This comparative approach is further limited to tasks with multiple reliable models, which may not always be available, especially in innovative or specialized applications.

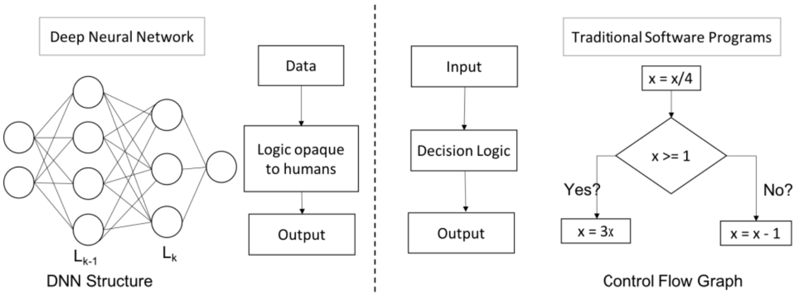

FIGURE 1.1: The internal logic of a deep neural network is opaque to humans, unlike the well-laid-out decision logic of traditional software programs [13]

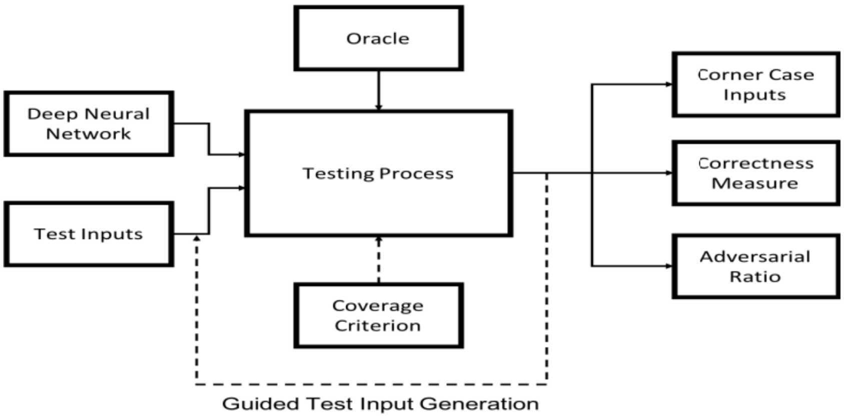

FIGURE 1.2: A high-level representation of most existing DNN testing methods [13]

1.1.2 Challenges of Deep Learning Models

The growing use of DNNs in safety-critical applications necessitates adequate testing to detect and correct any incorrect behavior for corner case inputs before deployment. DNNs lack an explicit control-flow structure, making it impossible to apply traditional software testing criteria such as code coverage.

- The input space is extremely large, making unguided simulations highly unlikely to find erroneous behavior.

1.1.3 Challenges in Testing of Deep Learning Models

Unlike traditional software, DNNs do not have a clear control-flow structure. They learn their decision policy through training on a large dataset, adjusting parameters gradually using several methods to achieve desired accuracy. Consequently, traditional software testing methods like functional coverage, branch coverage, etc., cannot be applied to DNNs, thereby challenging their use in safety-critical applications. Traditional software testing methods fail when applied to DNNs because the code for deep neural networks holds no information about the internal decision-making logic of a DNN.

DNN testing techniques aim to discover bugs by finding counterexamples that challenge the system's correctness or to establish confidence by rigorously evaluating the system with numerous test cases. These testing techniques are computationally less expensive and therefore can work with state-of-the-art DNNs. However, DL testing has some limitations:

- Standards available in the industry but **Lack of Logical Structure** and **System Specification**
- Heavily dependent on manual collections of test data under different conditions, which become expensive as the number of test conditions increases
- Existing coverage criteria are not detailed enough to notice subtle behaviors exhibited by DL systems

1.1.4 Problem Statement

Deep learning models are being more widely used in various applications, yet their reliability in practical applications remains a challenge.

1.1.5 Research Goal

This thesis aims to develop a systematic framework for evaluating local and global robustness in deep learning models. The goal is to provide a comprehensive error summary to improve model design and training, ensuring their reliability for real-world applications.

1.1.6 Research Questions

- How can we sample inputs efficiently?
- How can we design a comprehensive framework to test system robustness?
- How can we systematically evaluate the robustness both at local (property-specific) and global (overall system) levels within the framework?
- How can error summarization be employed to quantify the impacts on model robustness?

1.1.7 Thesis Contributions

This research makes the following key contributions to the field of deep learning robustness evaluation:

- We design an **end-to-end pipeline** for evaluating the correctness of the system.
- We propose a **conceptual framework** that quantifies both local and global correctness, with a formalized Bayesian probabilistic approach to verify system robustness.

- A novel **error summarization** approach which allows better identification of model weaknesses related to class and property.
- We perform all our **experiments** using publicly available deep learning models and the MNIST dataset.

1.1.8 Organization of the Thesis

The remainder of the thesis is organized as follows: related studies are presented in Chapter 2. The system model and proposed methodology are demonstrated in Chapter 3. Chapter 4 describes the simulation results of our proposed schemes. Finally, the findings of this work along with future directions are presented in Chapter **??**.

Chapter 2

Literature Review

2.1 Overview

This chapter provides a review of the literature related to deep neural networks, probabilistic logic programming, and their integration, particularly focusing on Problog and its extensions.

2.1.1 Deep Neural Networks and AI Systems

Deep neural networks (DNNs) mimic the structure of the human brain, consisting of millions of interconnected neurons. They extract high-level features from raw input using labeled training data without human interference.

Formally, a DNN is a function $f\colon \mathbb{R}^{s_0} \mapsto \mathbb{R}^{s_k}$ that takes as input a vector of size s_0 and produces a vector of size s_k. The function f is computed by composing k layers $L_1\colon \mathbb{R}^{s_0} \mapsto \mathbb{R}^{s_1}, \ldots, L_k\colon \mathbb{R}^{s_{k-1}} \mapsto \mathbb{R}^{s_k}$ as $f(x) = L_k(\cdots L_2(L_1(x))\cdots)$.

Each layer L_i typically implements a non-linear function. For instance, a *fully-connected* layer linearly transforms its input x_{i-1} as $Wx_{i-1} + b$, where $W \in \mathbb{R}^{s_i \times s_{i-1}}$ is the matrix of weights and $b \in \mathbb{R}^{s_i}$ is the bias vector. Then, it applies a non-linear activation function (e.g., sigmoid or Rectified Linear Unit (ReLU)) component-wise, generating the output vector x_i. The weights specify how its input neurons are connected to its output neurons and are known as *DNN parameters*. For more information about DNNs, we refer the reader to [1, 2, 3].

The objective of DNN training is to learn parameters during training to make accurate predictions on unseen data during real-world deployment.

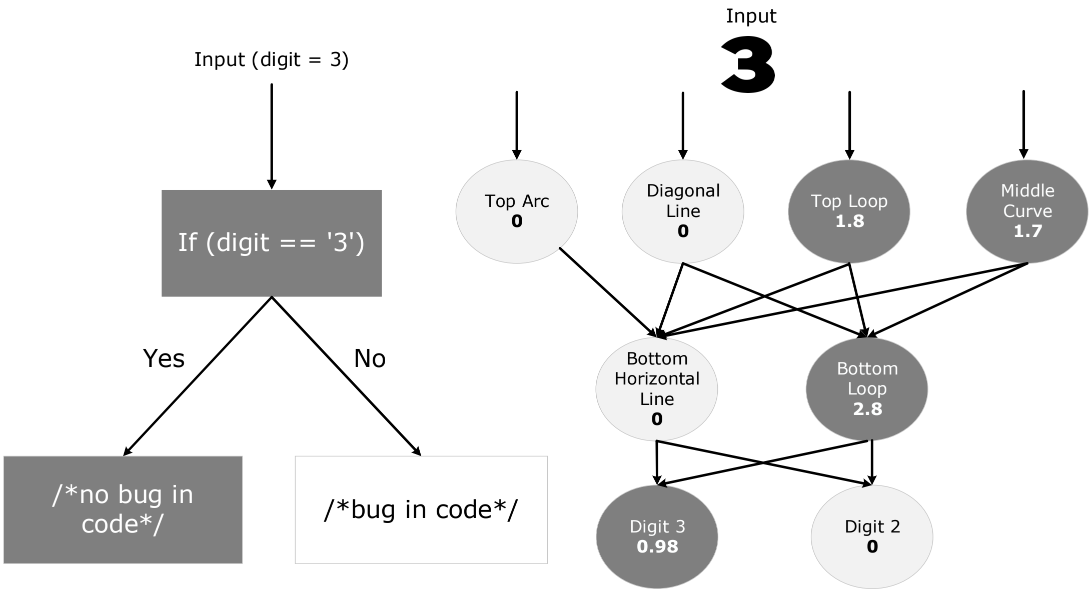

FIGURE 2.1: Comparison between program flows of a traditional program (left) and a neural network (right). The nodes in gray denote the corresponding basic blocks or neurons that participated while processing an input.

When the prediction task is classification, then s_k represents the number of classes. Assuming that $f(x) = (y_1, \ldots, y_{s_k})$, the *classification result* is $\operatorname*{argmax}_{i=1}^{s_k} y_i$, which is the index of the component with the highest probability y_i. By abuse of notation, sometimes we write $f(x) = c$ to denote the fact that x was classified as c. We also write $f(x)_c$ to refer to y_c which represents the probability of x being in class c.

By an *AI system*, we refer to any software system capable of performing complex tasks through the use of data, algorithms, and high computational power, which typically require human intelligence. These tasks include problem-solving, reasoning, decision-making, and natural language understanding.

Deep learning is a subset of AI that utilizes deep neural networks (DNNs) for complex pattern recognition. Some AI systems are solely based on DNN components, whereas *hybrid* AI systems combine DNNs with traditional software to produce the final output.

2.1.2 Robustness of DNNs

Deep neural networks (DNNs) are known for their lack of robustness. Research has shown DNNs to be vulnerable to two main categories of adversaries: *adversarial attacks* [4] and *image transformations* [5].

Let $\mathcal{A}$ denote an adversary. Each category can be described formally as follows:

Adversarial Attacks $\mathcal{A}_{\text{adv}}$

$(\textsc{adv})$ This involves the generation of perturbations δ such that $x' = x + \delta$ misleads the DNN f into making incorrect predictions, where x is the original input and x' is the perturbed input. Various methods under this category include:

- *Fast Gradient Sign Method (FGSM)*: For an input x and its true label y, FGSM generates $x' = x + \epsilon \cdot \text{sign}(\nabla_x J(x, y))$, where J is the loss function and ϵ is a small perturbation factor.
- *Basic Iterative Method (BIM)*: This extends FGSM by iteratively applying small perturbations: $x'^{(i+1)} = x'^{(i)} + \alpha \cdot \text{sign}(\nabla_x J(x'^{(i)}, y))$.
- *Carlini and Wagner (C&W) Attack*: Utilizes optimization to find δ that minimizes the L_p-norm while ensuring $f(x + \delta) \neq y$.
- *DeepFool*: Iteratively perturbs x by δ to move it across decision boundaries.
- *Jacobian-based Saliency Map Attack (JSMA)*: Manipulates specific features of x to achieve targeted misclassifications.

Image Transformations $\mathcal{A}_{\text{trans}}$

This involves modifying the input images in a manner that exploits the model's sensitivity to variations, potentially causing misclassifications. Various methods under this category include:

- *Noise Addition*: Introducing noise η such that $x' = x + \eta$. We consider *Gaussian* noise and *salt-and-pepper* noise.
- *Translation*: Shifting the image x by a vector t to obtain $x' = \text{translate}(x, t)$.
- *Scaling*: Resizing the image x by a factor s to get $x' = \text{scale}(x, s)$.
- *Shearing*: Applying a shear transformation to x resulting in $x' = \text{shear}(x, \theta)$.
- *Rotation*: Rotating the image x by an angle θ to produce $x' = \text{rotate}(x, \theta)$.
- *Contrast Adjustment*: Modifying the contrast of x, represented as $x' = \text{adjust\_contrast}(x, \alpha)$.
- *Brightness Change*: Altering the brightness of x, yielding $x' = \text{change\_brightness}(x, \beta)$.
- *Blurring*: Applying a blur effect to x, giving $x' = \text{blur}(x, k)$, where k is the kernel size.

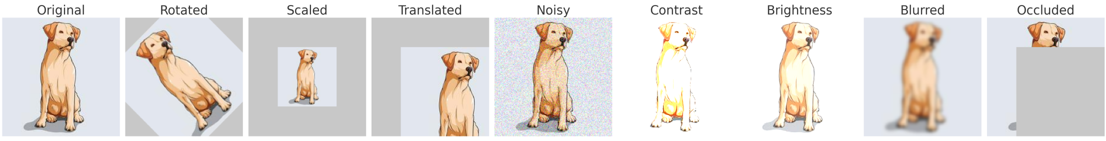

FIGURE 2.2: Image transformations for ...

- *Occlusion*: Partially hiding regions of x, leading to $x' = \text{occlude}(x, m)$, where m denotes the mask.

By analyzing various types of adversaries, our proposed comprehensive testing framework evaluates model robustness, providing probabilities of model effectiveness against each adversary and assessing accuracy under adversarial conditions across different classes within any dataset.

2.1.3 DNN Testing Techniques

The development of DNNs is significantly different from traditional software. While developers explicitly define logic in traditional software, DNNs learn logic rules from raw data. Developers shape these rules by modifying the training data, selecting features, and designing the DNN architecture, such as the number of neurons and layers.

Since the logic of a DNN is non-transparent [6], identifying the reasons behind its erroneous behavior is challenging. Therefore, testing and correcting its errors are crucial, particularly in safety-critical systems. Next, we briefly introduce two major DNN testing techniques: coverage criteria and test-case generation.

Coverage Criteria

In traditional software testing, coverage criteria measure how thoroughly software is tested. In DNNs, coverage might not directly apply to lines of code but rather to the input space or the variety of data the model can effectively handle or provide predictions for.

Neuron coverage (NC) [6] is the first coverage metric proposed in the literature to test DNNs. It is defined as the ratio of neurons activated by a test input to the total number of neurons in the model, where a neuron is activated when its activation value exceeds a predefined threshold.

Ma et al. [7] proposed a variety of coverage metrics, including K-multisection neuron coverage (KMNC), Neuron boundary coverage (NBC), and Strong neuron activation coverage (SNAC). KMNC calculates coverage by dividing the interval between lower and upper bounds into k-bins and measuring the number of bins activated by the test inputs. NBC measures the ratio of corner case regions covered by test inputs, with corner cases defined as activation values below or above those observed during training. SNAC similarly measures how many upper corner cases, defined as activation values above the training range, are covered by test inputs.

Modified Condition/Decision Coverage (MC/DC) [8] captures causal changes in test inputs based on the sign and value change of a neuron's activation.

Likelihood-based Surprise Adequacy (LSA) uses Kernel Density Estimation (KDE) to estimate the likelihood of a test input during the training phase, prioritizing inputs with higher LSA scores as they are closer to classification boundaries. Distance-based Surprise Adequacy (DSA) is an alternative to LSA that uses the distance between activation traces of new test inputs and those observed during training [9].

DNN Test-case Generation

Test-case generation methods are influenced by traditional software testing methods like fuzz testing, metamorphic testing, and symbolic execution. In the following sections, we will explore the current state of the art in DNN test generation.

DeepXplore [6] is a whitebox test-case generation method that checks how different DNNs behave using domain-specific rules on inputs. It uses multiple models trained on the same data to find differences in their prediction. It aims to jointly optimize neuron coverage and different predictions between models, using gradient ascent for test generation.

DeepTest [5] focuses on generating test inputs for autonomous cars by applying domain-specific rules on seed inputs. It uses a greedy search method based on the NC metric to create effective test cases.

Adapting traditional fuzzing techniques for DNN test-case generation includes methods like DLFuzz [10] and TensorFuzz [11]. DLFuzz generates adversarial inputs based on NC, akin to DeepXplore, but does not require multiple models and uses constraints to keep new inputs similar to originals. TensorFuzz employs coverage-guided testing to uncover numerical issues and discrepancies in DNNs and their quantized versions.

DeepConcolic [12] employs a concolic testing approach to generate adversarial inputs for DNN testing. It combines symbolic execution with concrete execution path information to meet coverage criteria, supporting both NC and MC/DC criteria.

Traditional techniques are simple, failing to capture the full complexity and precision of model behaviors. Exploring all possible behaviors of a model is nearly impossible due to the vast number of paths to consider. These metrics also often overlook the detailed interactions within and between layers of the model. Defining and testing all necessary decision boundaries, especially in complex models, is a daunting task. Many existing metrics do not provide clear directions for improving the model, leaving you without actionable insights. Scalability and adaptability are other major issues. Many criteria are not scalable or adaptable across diverse model architectures.

In this thesis, we address these issues and design a systematic testing framework for DNNs.

2.1.4 Probabilistic Logic Programming (PLP)

Probabilistic Logic Programming (PLP) integrates probabilistic reasoning with the flexibility and expressiveness of logic programming. Traditional logic programming paradigms are deterministic, where each statement is either true or false. However, real-world scenarios often involve inherent uncertainties which deterministic logic cannot adequately handle. PLP addresses this limitation by allowing the representation of uncertainties directly within the logic framework, thereby enabling more nuanced and accurate modeling of complex domains [29, 30].

PLP has been explored extensively in various domains, including artificial intelligence, bioinformatics, and robotics. Sato and Kameya's introduction of PRISM [23] was a significant milestone, combining statistical modeling with logic programming. Subsequent work by Koller and Friedman [24] on probabilistic graphical models further influenced PLP frameworks. Various approaches to PLP, such as Logic Programs with Annotated Disjunctions (LPADs) [25], ProbLog [26], Probabilistic Horn Abduction (PHA) [27], Independent Choice Logic (ICL) [28], and PRISM [29], have been developed, each leveraging the distribution semantics introduced by Sato [29].

Among these, ProbLog has gained prominence due to its simplicity and expressive power, making it a preferred choice for many applications [30]. ProbLog extends a logic program with probabilistic facts, defining a probability distribution over possible worlds (sets of facts). The probability of a query is computed by marginalizing over the probabilities of the worlds where the query is true.

The need for explainable AI (XAI) has become increasingly important, particularly in domains where decisions must be transparent and interpretable, such as medical diagnosis and finance. Various approaches to XAI emphasize model interpretability and the generation of comprehensible explanations for model predictions [32]. PLP, with its ability to generate explanations as part of its reasoning process, fits well within the XAI paradigm. Vidal [31] proposes a novel approach where explanations in PLP are represented as programs generated from a given query through unfolding-like transformations. This approach preserves the causal structure of inferences and ensures minimality by excluding irrelevant information. Such explanations can be parameterized to hide uninteresting details, making them comprehensible to non-expert users.

Despite the extensive research in PLP and its applications, its use in testing DNNs is still very new. DNNs are powerful but often act like "black boxes," making it hard to understand and trust their predictions. This thesis presents a new way to use PLP for testing DNNs, an area that has not been explored before.

By using PLP, we can turn the probabilistic outputs of DNNs into probabilistic facts and rules. This helps in reasoning about uncertainties in the model's predictions and generating clear explanations for each prediction. This approach makes it easier to understand why a model made a certain prediction, giving insights into the model's decision-making process. PLP can also help in debugging DNNs by identifying and fixing inconsistencies or unexpected behaviors in the predictions. By examining the probabilistic rules and their explanations, we can find areas for improvement more effectively. Additionally, PLP allows us to simulate different scenarios by changing probabilistic facts, which is useful for testing how robust the DNNs are under various conditions. This innovative method opens up new opportunities for research and practical applications, making deep learning systems more transparent, reliable, and trustworthy.

Chapter 3

Proposed Framework

3.1 Proposed Approach

This section introduces our comprehensive approach to evaluating AI system performance, summarized in Figure 3.1. It takes as input the description of an AI system and a set of relevant specifications against which the AI system should be checked. The framework itself has four main components:

(i) *Sampling*: Choosing the original inputs for the AI system.

(ii) *Testcase Generation*: Generating specific test cases from the sampled inputs according to the specifications.

(iii) *Validation*: Checking the behavior of the AI system on the generated test cases, which can be fully-fledged *verification* or more lightweight *testing*.

(iv) *Error Summarization*: Quantifying the performance of the AI system in terms of its global/local robustness/correctness.

We focus on AI systems with DNN components performing classification tasks. We assume that an AI system $\mathcal{S}$ is a pair $(\mathcal{F}, \mathcal{D})$ where:

- $\mathcal{F}$ is the functional unit consisting of n DNN *classifiers* $f_1, \ldots, f_n$ and a symbolic (software) component ω such that given an input $\boldsymbol{x} = (x_1, \ldots, x_n)$, the *output* $\mathcal{F}(\boldsymbol{x})$ is defined as $\omega(f_1(x_1), \ldots, f_n(x_n))$.

- $\mathcal{D}$ (for dataset) is a structure that describes valid inputs for each classifier f_i and the corresponding correct outputs (i.e., labels). In particular, $\mathcal{D}.next_i(param)$

returns a valid input x_i for f_i, given the parameter $param$. Moreover, $\mathcal{D}.N_i$ is the number of distinct class labels for each classifier f_i.

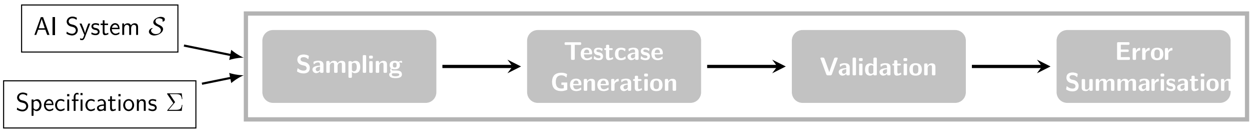

FIGURE 3.1: Overview of the Proposed Framework

Example 3.1. *An instance of a simple AI system is an* MNIST Digit Adder $\mathcal{S}_{\mathit{MNIST}} = (\mathcal{F}, \mathcal{D})$*, where* $\mathcal{F} = (\{f_{\mathit{mnist}}\}, +)$*,* f_{mnist} *is an MNIST Digit classifier and* $\mathcal{F}$ *takes as input two MNIST Digit images, recognizes the digits in the images, and computes their sum, i.e.,* $\mathcal{F}(x_1, x_2) = f_{\mathit{mnist}}(x_1) + f_{\mathit{mnist}}(x_2)$*.* $\mathcal{D}$ *is the testing dataset for MNIST Digit consisting of 10,000 labeled images.*

We assume that *specifications* Σ is a pair (P, V), where P is a set of perturbations against which we are characterizing the behavior of $\mathcal{S}$. Each perturbation comes with parameters to instantiate the set of all possible perturbations. V is a validation flag, where if $V = t$, then we do testing, and if $V = v$, we do verification.

Example 3.2. *To evaluate the correctness of the MNIST Digit Adder* $\mathcal{S}_{\mathit{MNIST}}$*, we define the specifications* $\Sigma = (P, V)$ *as follows:* $P = \{\textsc{gau}(0, 0.1), \textsc{sap}(200 : 255, 0 : 5, 0.2), \textsc{rot}(3, 30, 3)\}$ *and* $V = t$*. Here,* $\textsc{gau}(0, 0.1)$ *specifies Gaussian noise with a mean of* 0 *and standard deviation of* 0.1*,* $\textsc{sap}(200 : 255, 0 : 5, 0.2)$ *specifies salt and pepper noise, where 10% of pixels are bleached up to values 200 to 255 and 10% of pixels are darkened to values between 0 and 5, and* $\textsc{rot}(3, 30, 3)$ *specifies the set of rotations with the minimum rotation angle of 3, maximum of 30, and the step size of 3.*

3.2 Sampling

The sampling process involves a random but balanced choice of samples from each class, focusing exclusively on instances that the model has correctly predicted. Sampling happens independently for each classifier f_i. To ensure a representative and fair distribution of data across all classes, we sample the same number of instances from each class. The full sample S_i for classifier f_i, $i = 1, \ldots, n$, is computed as:

$$S_i = \bigcup_{c=1}^{\mathcal{D}.N_i} S_i^c \tag{3.1}$$

where S_i^c is a subset of the correctly classified samples for a class c consisting of M_i (the number of samples for each class specific to f_i) elements:

$$S_i^c = \Big\{x = \mathcal{D}.next_i(class = c) \mid f_i(x) = c\Big\}|_{M_i}$$

Note that each sample is obtained by a call to the method $\mathcal{D}.next_i$.

Example 3.3. *Recall the MNIST Digit Adder system $\mathcal{S}_{\mathit{MNIST}}$ from Example 3.1. When sampling, we make sure to select only correctly classified samples. For each digit c (0 to 9), we randomly sample 100 images, i.e., $samp_1(\mathcal{D}, c, 100)$ contains 100 inputs x such that $f_{\mathit{mnist}}(x) = c$. In total, S_1 contains 1000 samples for all 10 classes.*

3.3 Test Case Generation

The test case generation process aims to create test cases based on the given specifications to evaluate the correctness/robustness of the AI system.

Let S_i^c be the set of samples produced in the sampling step for the classifier f_i and a class c. For each perturbation $p \in P$, we generate a set $\mathcal{T}_p^c$ of test cases. Specifically, for each sample $x \in S$ we produce $testcases(p, c, x)$ according to p. Then $\mathcal{T}_p^c = \bigcup_{x \in S} testcases(p, c, x)$.

Example 3.4. *To generate test cases for the MNIST Digit Adder $\mathcal{S}_{\mathit{MNIST}}$, we use the specifications defined in Example 2, which include noise and rotation perturbations. Let S be the set of sampled images obtained in Example 3. For each pair of images $(x_1, x_2) \in S$, we define the following test cases:*

- ***Rotation**: For a given angle θ, generate the perturbed images $x_1' = \mathit{rotate}(x_1, \theta)$ and $x_2' = \mathit{rotate}(x_2, \theta)$. The test case is then: $\mathcal{T}_{\mathit{rotation}} = \{(x_1', x_2') \mid x_1', x_2' \in \mathit{rotate}(S, \theta)\}$*
- ***Noise**: For a given mean μ and standard deviation σ, generate the perturbed images $x_1'' = \mathit{noise}(x_1, \mu, \sigma)$ and $x_2'' = \mathit{noise}(x_2, \mu, \sigma)$. The test case is then: $\mathcal{T}_{\mathit{noise}} = \{(x_1'', x_2'') \mid x_1'', x_2'' \in \mathit{noise}(S, \mu, \sigma)\}$*

The overall set of test cases $\mathcal{T}$ is the union of the individual test cases: $\mathcal{T} = \mathcal{T}_{\mathit{rotation}} \cup \mathcal{T}_{\mathit{noise}}$

3.4 Validation

The validation process aims to evaluate the correctness and robustness of the AI system under various perturbations.

Fix a perturbation $p \in P$ and a class c. For every test case in $\mathcal{T}_p^c$, we store the results in the form:

$$\text{Raw}_{p,c} = \left\{ \left(\text{query}(f_i, x, c), f_i(x)_c \right) \mid x \in \mathcal{T}_p^c \right\}$$

After generating test cases, measure the AI subsystem's confidence for each class under each type of property.

3.4.1 Local Robustness

Local robustness involves evaluating the AI subsystem's performance on individual images subjected to various transformations. For each image x from a set of samples S_i^c, the AI subsystem produces a confidence score $f_i(x)$ representing its certainty in recognizing the class c. The local robustness for a transformation T applied to an image x is defined as:

$$\text{Local Robustness}(x, T) = f_i(T(x))$$

where T can be any transformation such as noise addition, rotation, brightness adjustment, occlusion, or scaling. Each transformation is evaluated to determine its impact on the confidence score.

To quantify local robustness, we compute the accuracy for each transformation applied to the images of a specific class:

$$\text{Local Robustness}(c, T) = \frac{1}{|S_i^c|} \sum_{x \in S_i^c} \mathbb{I}[f_i(T(x)) = c]$$

where: - Local Robustness(c, T) is the accuracy of the classifier f_i for class c under transformation T. - $\mathbb{I}[f_i(T(x)) = c]$ is an indicator function that is 1 if the classifier correctly predicts the class c for the transformed image $T(x)$, and 0 otherwise. - $|S_i^c|$ is the number of correctly classified images in class c.

Example 3.5. *Consider the class $c = 3$ and the transformation T being a rotation by 25 degrees. If we have three images x_1, x_2, x_3 from class c, and the model correctly classifies x_1 and x_2 but misclassifies x_3, the local robustness is computed as follows:*

$$\textit{Local Robustness}(3, \textit{rotation}) = \frac{1}{3}(\mathbb{I}[f_i(\textit{rotate}(x_1, 25)) = 3] +$$
$$\mathbb{I}[f_i(\textit{rotate}(x_2, 25)) = 3] + \mathbb{I}[f_i(\textit{rotate}(x_3, 25)) = 3])$$

If the indicator values are 1, 1, and 0 respectively, the local robustness would be:

$$\textit{Local Robustness}(3, \textit{rotation}) = \frac{1}{3}(1 + 1 + 0) = \frac{2}{3} \approx 0.67$$

3.4.2 Global Robustness

Global robustness evaluates the AI subsystem's performance across multiple images and transformations. It is an aggregate measure of how well the AI system performs under various properties on a set of images S_i^c.

For a given transformation T and a set of images S_i^c, the global robustness is defined as:

$$\text{Global Robustness}(S_i^c, T) = \frac{1}{|S_i^c|} \sum_{x \in S_i^c} \mathbb{I}[f_i(T(x)) = c]$$

This measures the average confidence score for the AI subsystem over the entire dataset when subjected to a particular transformation.

To quantify global robustness, we compute the expected confidence score over all transformations and images. Depending on whether the relationship is AND or OR, the formulas vary:

AND Relationship:

For an AND relationship between transformations, the global robustness is calculated as:

$$P(\text{Property 1} \cap \text{Property 2}) = P(\text{Property 1}) \times P(\text{Property 2})$$

OR Relationship:

For an OR relationship between transformations, the global robustness is calculated as:

$$P(\text{Property 1} \cup \text{Property 2}) = P(\text{Property 1}) + P(\text{Property 2}) - P(\text{Property 1} \cap \text{Property 2})$$

Example 3.6. *Consider a pair of images (x_1, x_2) representing the digits '3' and '5'. Let the transformation T be rotation. If the model's confidence scores for these transformations are as follows:*

$$f_i(\textit{rotation}(x_1)) = 0.78,$$
$$f_i(\textit{rotation}(x_2)) = 0.85,$$

For the AND relationship, the global correctness for the pair is computed as follows:

$$P(\textit{Global Robustness}_{\textit{AND}}) = 0.78 \times 0.85$$

For the OR relationship, the global correctness for the pair is computed as follows:

$$P(\textit{Global Robustness}_{\textit{OR}}) = 0.78 + 0.85 - (0.78 \times 0.85)$$

(Note: The complete OR relationship formula requires including all images and transformations as per the OR formula.)

3.5 Use Cases and Examples

To illustrate the application of ProbLog for global robustness in real-world scenarios, we present the following use cases:

3.5.1 Use Case 1: Handwritten Digit Recognition

Scenario: Consider an AI system designed to recognize handwritten digits, such as the MNIST dataset. The system is evaluated under various transformations, including noise addition, rotation, and brightness adjustment. The goal is to determine the global robustness of the system in recognizing digit pairs correctly under these properties.

The tables below provide the probabilities for correctly recognizing digit pairs under different conditions (AND and OR relationships) for an MNIST 2-digit addition system. Each world represents a different combination of transformations applied to the digits.

Explanation: The table shows the global correctness probabilities for pairs of digits under various transformation conditions. Each row represents a different combination of transformations applied to the two digits in the pair: - AND Probability: The probability that both digits are correctly recognized under the specified transformations. - OR

Table 3.1: Specification Probabilities for MNIST 2-Digit Addition Under Different Transformations

World	Conditions	Probability Expression (AND)	Probability (AND)	Probability Expression (OR)	Probability (OR)
w_1	{noise(0), noise(1)}	$0.85 \cdot 0.8$	0.68	$0.85 + 0.8 - (0.85 \cdot 0.8)$	0.97
w_2	{noise(0), correct(1)}	$0.85 \cdot 0.9$	0.765	$0.85 + 0.9 - (0.85 \cdot 0.9)$	0.985
w_3	{correct(0), noise(1)}	$0.9 \cdot 0.8$	0.72	$0.9 + 0.8 - (0.9 \cdot 0.8)$	0.98
w_4	{rotation(0), correct(1)}	$0.88 \cdot 0.9$	0.792	$0.88 + 0.9 - (0.88 \cdot 0.9)$	0.992
w_5	{correct(0), rotation(1)}	$0.9 \cdot 0.77$	0.693	$0.9 + 0.77 - (0.9 \cdot 0.77)$	0.977
w_6	{rotation(0), rotation(1)}	$0.88 \cdot 0.77$	0.6776	$0.88 + 0.77 - (0.88 \cdot 0.77)$	0.9696
w_7	{noise(0), rotation(1)}	$0.85 \cdot 0.77$	0.6545	$0.85 + 0.77 - (0.85 \cdot 0.77)$	0.9655
w_8	{rotation(0), noise(1)}	$0.88 \cdot 0.8$	0.704	$0.88 + 0.8 - (0.88 \cdot 0.8)$	0.976
w_9	{correct(0), correct(1)}	$0.9 \cdot 0.9$	0.81	$0.9 + 0.9 - (0.9 \cdot 0.9)$	0.99

Probability: The probability that at least one of the digits is correctly recognized under the specified transformations.

For example, in world w_1, both digits are subjected to noise, leading to an AND probability of 0.68 and an OR probability of 0.97.

Problog Code:

```
% Define probabilities for digit 0 under different transformations
0.9::noise_0. % Digit 0 correctly predicted with 90% probability under noise
0.85::brightness_0. % Digit 0 correctly predicted with 85% probability under brightness
0.88::rotation_0. % Digit 0 correctly predicted with 88% probability under rotation

% Define probabilities for digit 1 under different transformations
0.8::noise_1. % Digit 1 correctly predicted with 80% probability under noise
0.75::brightness_1. % Digit 1 correctly predicted with 75% probability under brightness
0.77::rotation_1. % Digit 1 correctly predicted with 77% probability under rotation

% Define rules for correct prediction under each transformation for digit 0
correct_noise_0 :- noise_0.
correct_brightness_0 :- brightness_0.
correct_rotation_0 :- rotation_0.

% Define rules for correct prediction under each transformation for digit 1
correct_noise_1 :- noise_1.
correct_brightness_1 :- brightness_1.
correct_rotation_1 :- rotation_1.

% Define rules for incorrect prediction under each transformation for digit 0
wrong_noise_0 :- +correct_noise_0.
wrong_brightness_0 :- +correct_brightness_0.
wrong_rotation_0 :- +correct_rotation_0.

% Define rules for incorrect prediction under each transformation for digit 1
wrong_noise_1 :- +correct_noise_1.
wrong_brightness_1 :- +correct_brightness_1.
wrong_rotation_1 :- +correct_rotation_1.

% Define rules for correct prediction of both digits under noise
pair_correct_noise_0_1 :- correct_noise_0, correct_noise_1.
% Define rules for incorrect prediction of both digits under noise
pair_wrong_noise_0_1 :- wrong_noise_0, wrong_noise_1.
```

```
% Define global correctness based on either both correct or both incorrect under noise
global_correct_noise_0_1 :- pair_correct_noise_0_1; pair_wrong_noise_0_1.

% Query the global correctness under noise
query(global_correct_noise_0_1).
```

FIGURE 3.2: Problog code snippet for evaluating handwritten digit recognition under noise, brightness, and rotation transformations.

Explanation: In this scenario, we are interested in the global correctness of recognizing pairs of digits (0 and 1) under different transformations. The ProbLog code models the local robustness probabilities for each transformation and combines them to evaluate the global correctness.

3.5.2 Use Case 2: Autonomous Vehicle Perception

Scenario: An AI system used in autonomous vehicles must reliably detect objects such as vehicles under various weather conditions (rain, sand, fog, and snow). The goal is to evaluate the system's robustness in identifying these objects correctly under these weather conditions.

```
% Probabilities for Vehicle Detection under Different Weather Conditions
0.75::rain_vehicle. % Vehicle correctly detected with 75% probability under rain
0.55::fog_vehicle. % Vehicle correctly detected with 55% probability under fog
0.7::snow_vehicle. % Vehicle correctly detected with 70% probability under snow

% Correct Detection Rules for Vehicle
correct_rain_vehicle :- rain_vehicle.
correct_fog_vehicle :- fog_vehicle.
correct_snow_vehicle :- snow_vehicle.

% Incorrect Detection Rules for Vehicle
wrong_rain_vehicle :- +correct_rain_vehicle.
wrong_fog_vehicle :- +correct_fog_vehicle.
wrong_snow_vehicle :- +correct_snow_vehicle.

% AND conditions for Vehicle Detection under all weather conditions
global_correct_vehicle_and :- correct_rain_vehicle, correct_fog_vehicle, correct_snow_vehicle.

% OR conditions for Vehicle Detection under any weather condition
global_correct_vehicle_or :- correct_rain_vehicle; correct_fog_vehicle; correct_snow_vehicle.

% Mixed conditions (AND & OR) for Vehicle Detection
global_correct_mixed_vehicle :- correct_rain_vehicle, (correct_fog_vehicle; correct_snow_vehicle).

% Queries for Global Correctness
query(global_correct_vehicle_and).
query(global_correct_vehicle_or).
query(global_correct_mixed_vehicle).
```

FIGURE 3.3: Problog code snippet for evaluating vehicle detection under different weather conditions.

Explanation: The ProbLog code assesses the global robustness of the AI system in detecting objects (vehicles) under individual and combined weather conditions. This ensures that the system can reliably perform in diverse environmental scenarios.

TABLE 3.2: Specification Probabilities (AND) for Vehicle Detection Under Different Weather Conditions

$$P(A \cap B \cap C) = P(A) \times P(B) \times P(C)$$

World	Conditions	Probability Expression (AND)	Probability (AND)
w_1	{rain, fog}	0.75×0.55	0.4125
w_2	{rain, snow}	0.75×0.7	0.525
w_3	{rain, sand}	0.75×0.6	0.45
w_4	{fog, snow}	0.55×0.7	0.385
w_5	{fog, sand}	0.55×0.6	0.33
w_6	{snow, sand}	0.7×0.6	0.42
w_7	{rain, fog, snow}	$0.75 \times 0.55 \times 0.7$	0.28875
w_8	{rain, fog, sand}	$0.75 \times 0.55 \times 0.6$	0.2475
w_9	{rain, snow, sand}	$0.75 \times 0.7 \times 0.6$	0.315
w_{10}	{fog, snow, sand}	$0.55 \times 0.7 \times 0.6$	0.231

Explanation: This table shows the global correctness probabilities for detecting vehicles under different combinations of weather conditions using AND relationships. Each row represents a different combination of weather conditions applied to the detection scenario: - AND Probability: The probability that the vehicle is correctly detected under all specified weather conditions.

For example, in world w_1, the vehicle detection system is subjected to rain and fog, leading to an AND probability of 0.4125.

TABLE 3.3: Specification Probabilities (OR) for Vehicle Detection Under Different Weather Conditions

$$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$$

World	Conditions	Probability Expression (OR)	Probability (OR)
w_1	{rain; fog}	$0.75 + 0.55 - (0.75 \times 0.55)$	0.8875
w_2	{rain; snow}	$0.75 + 0.7 - (0.75 \times 0.7)$	0.925
w_3	{rain; sand}	$0.75 + 0.6 - (0.75 \times 0.6)$	0.9
w_4	{fog; snow}	$0.55 + 0.7 - (0.55 \times 0.7)$	0.835
w_5	{fog; sand}	$0.55 + 0.6 - (0.55 \times 0.6)$	0.82
w_6	{snow; sand}	$0.7 + 0.6 - (0.7 \times 0.6)$	0.88
w_7	{rain; fog; snow}	$0.75 + 0.55 + 0.7 - (0.75 \times 0.55) - (0.75 \times 0.7) - (0.55 \times 0.7) + (0.75 \times 0.55 \times 0.7)$	0.966625
w_8	{rain; fog; sand}	$0.75 + 0.55 + 0.6 - (0.75 \times 0.55) - (0.75 \times 0.6) - (0.55 \times 0.6) + (0.75 \times 0.55 \times 0.6)$	0.95125
w_9	{rain; snow; sand}	$0.75 + 0.7 + 0.6 - (0.75 \times 0.7) - (0.75 \times 0.6) - (0.7 \times 0.6) + (0.75 \times 0.7 \times 0.6)$	0.967
w_{10}	{fog; snow; sand}	$0.55 + 0.7 + 0.6 - (0.55 \times 0.7) - (0.55 \times 0.6) - (0.7 \times 0.6) + (0.55 \times 0.7 \times 0.6)$	0.938

Explanation: This table shows the global correctness probabilities for detecting vehicles under different combinations of weather conditions using OR relationships. Each row represents a different combination of weather conditions applied to the detection scenario: - OR Probability: The probability that the vehicle is correctly detected under at least one of the specified weather conditions.

For example, in world w_1, the vehicle detection system is subjected to rain and fog, leading to an OR probability of 0.8875.

3.6 Error Summarization

Error summarization involves evaluating the performance of the AI system by identifying and quantifying errors. We use Bayesian Network-based Coverage Metrics to assess both local and global coverage. Two testing coverage metrics are defined in Figure **??**: local coverage (LC) and global coverage (GC).

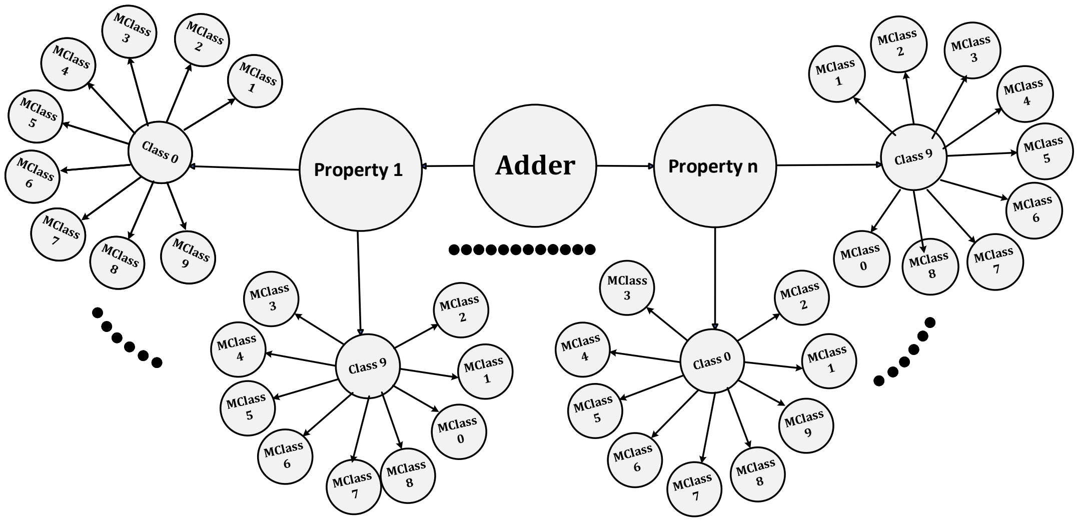

FIGURE 3.4: Error Summarization

3.6.1 Algorithm for Evaluating Model Robustness Using ProbLog

Algorithm 1 Evaluating Model Robustness Using ProbLog

```
Require: $M$: Pre-trained model
Require: $D$: Dataset
Require: $N$: Number of samples per class
Require: $P$: Set of properties
Ensure: $G$: Global correctness values for each specification
Load and Preprocess Data
$(X_{\text{train}}, y_{\text{train}}), (X_{\text{test}}, y_{\text{test}}) \leftarrow$ load_data$(D)$
$X_{\text{test}} \leftarrow$ preprocess_data$(X_{\text{test}})$
$\hat{Y} \leftarrow M.$predict$(X_{\text{test}})$
$\hat{y} \leftarrow$ extract_predictions$(\hat{Y})$
Determine Number of Classes
$C \leftarrow$ num_classes$(y_{\text{test}})$ {Number of unique classes in the dataset}
Select Correctly Classified Samples
correct_samples $\leftarrow \{\}$
for $c = 0$ to $C - 1$ do
    correct_samples$[c] \leftarrow$ select_correct_samples$(X_{\text{test}}, y_{\text{test}}, \hat{y}, c, N)$
end for
Define Transformation Functions
define_transformations()
Generate Test Cases
for $c = 0$ to $C - 1$ do
    for all $x \in$ correct_samples$[c]$ do
        for all $T \in P$ do
            generate_test_cases$(x, T)$
        end for
    end for
end for
Evaluate Model on Transformations
for $c = 0$ to $C - 1$ do
    for all property $\in P$ do
        $L_{c,\text{property}} \leftarrow$ compute_accuracy$(M, \text{property}, c)$
    end for
end for
Specification Definitions:
Specification1: $P$(Property1 $\cap$ Property2) $=$ $P$(Property1) $\times$ $P$(Property2) {AND relationship}
Specification2: $P$(Property1 $\cup$ Property2) $=$ $P$(Property1) $+$ $P$(Property2) $-$ $P$(Property1 $\cap$ Property2) {OR relationship}
Specification3: Custom definitions
...
Generate and Evaluate ProbLog Code for Each Specification
Initialize $G$ as an empty list
for spec $= 1$ to num_specs do
    ProbLog Code$_{\text{spec}} \leftarrow$ generate_problog_code$(L_{c,\text{property}}, \text{spec})$
    $G_{\text{spec}} \leftarrow$ evaluate_problog(ProbLog Code$_{\text{spec}}$)
    Append $G_{\text{spec}}$ to $G$
end for
return $G$
```

Chapter 4

Simulations and Results

4.1 Datasets

4.1.1 MNIST Dataset

The MNIST dataset is a widely recognized benchmark in the field of deep neural networks (DNNs), comprising 70,000 grayscale images of handwritten digits. These images are split into 60,000 training samples and 10,000 testing samples, each of size 28x28 pixels. The dataset includes labels for each digit from 0 to 9, making it a total of 10 classes. This dataset is extensively used for evaluating the performance of various DNN algorithms due to its simplicity and the well-established baseline results it offers. To ensure the correctness our model, we selected 100 correctly classified samples from each class. This selection process involved comparing the model's predictions with the actual labels and choosing only those samples where the predictions were correct. This resulted in a balanced subset of 1,000 samples, with 100 samples from each of the 10 classes.

4.1.2 DAWN Dataset

The DAWN (Vehicle Detection in Adverse Weather Nature Dataset) dataset focuses on vehicle detection under adverse weather conditions, providing a diverse set of real-traffic images categorized into four weather conditions: fog, snow, rain, and sandstorms. Initially, the class distribution in the training set was imbalanced, with the following counts: 258 for label 1, 240 for label 0, 163 for label 3, and 160 for label 2.

To address this imbalance, we resampled the training set to ensure an equal number of samples for each class. This resulted in a balanced training set with 258 samples for each label. This balancing step was crucial for fair training and evaluation of the model, ensuring that no class was overrepresented or underrepresented.

For the testing set, we selected 100 samples from each class, ensuring a balanced evaluation dataset. This selection process was critical to accurately assess the model's performance across different classes and adverse weather conditions.

4.2 Use Case 1: Handwritten Digit Recognition

4.2.1 Local Correctness Evaluation

In this section, we evaluate the local correctness of the model under different transformations, namely rotation, noise, and brightness. Each transformation is applied with specific parameters to simulate real-world variations. Images are rotated by 25 degrees to evaluate the model's performance under rotation. Gaussian noise with a noise factor of 0.2 is added to the images to test the model's robustness to noisy conditions. The brightness of the images is increased by a factor of 0.3 to simulate different lighting conditions.

The local correctness graphs provide insights into the model's robustness under these transformations for each digit class. The model exhibited high performance under rotation (0.99) and brightness (1.00) for Class 0, with slightly reduced accuracy under noise (0.81). For Class 1, the model maintained consistently high accuracy across all transformations: rotation (0.99), noise (0.99), and brightness (1.00). However, the model struggled more with rotations (0.76) and noise (0.79) for Class 2 compared to brightness (1.00). Similarly, for Class 3, the model showed lower accuracy for rotations (0.82) and noise (0.78), but performed perfectly under brightness (1.00).

Class 4 demonstrated high accuracy across all properties, with rotation (0.91), noise (0.86), and brightness (1.00). Class 5 maintained high performance under all transformations: rotation (0.89), noise (0.99), and brightness (1.00). The model showed noticeable difficulty with noise (0.64) for Class 6, while maintaining high accuracy for rotation (0.87) and brightness (0.99). For Class 7, high accuracy was observed under brightness (1.00), with reduced performance under rotation (0.75) and noise (0.86). The model faced significant challenges under noise (0.02) for Class 8, but maintained good performance for rotation (0.68) and brightness (0.99). Finally, for Class 9, the model showed moderate performance under noise (0.51), and high accuracy under rotation (0.93) and brightness (0.99).

Overall, the model demonstrates strong robustness to brightness changes across all classes, frequently achieving perfect accuracy. However, performance under noise is highly variable, with some classes, such as Class 8, experiencing a dramatic drop in accuracy. While the model generally handles rotations well, there remains room for improvement, particularly in addressing noise-induced challenges. This analysis highlights the necessity for targeted training to enhance the model's robustness, especially under noisy conditions for specific classes.

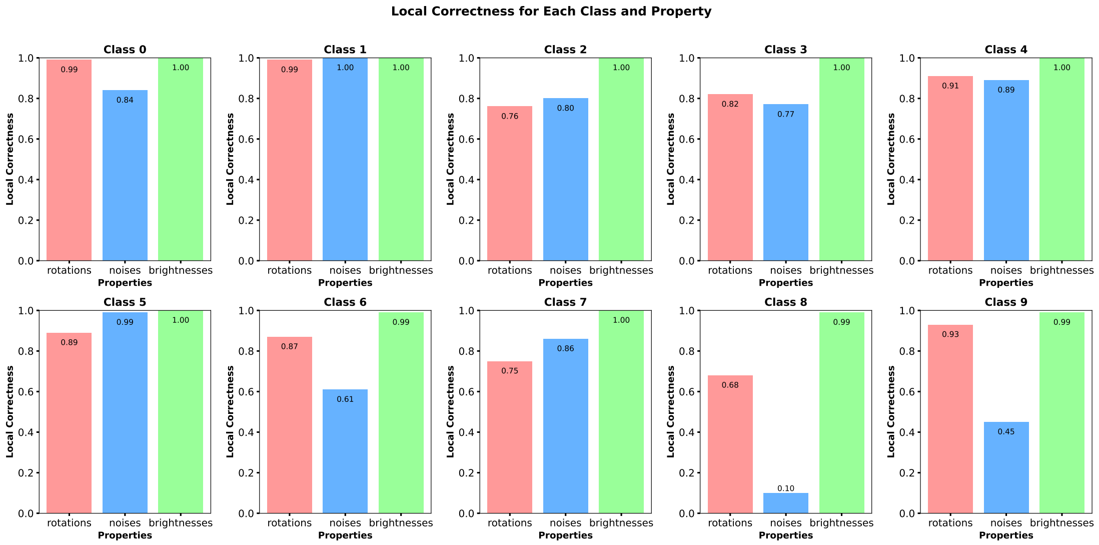

FIGURE 4.1: Local Correctness for each Class and Property

4.2.2 Global Correctness Evaluation

The following figures illustrate the global correctness of various pairs of digits under different transformations: noise, brightness adjustment, and rotation. The final global correctness for each transformation is also depicted.
In my experiment, I checked the following specifications:

$$\begin{aligned} global\_noise &\leftrightarrow (noise\_0\_0 \wedge noise\_0\_1 \wedge noise\_0\_2 \wedge \ldots \wedge noise\_9\_9) \\ global\_brightness &\leftrightarrow (brightness\_0\_0 \wedge brightness\_0\_1 \wedge brightness\_0\_2 \wedge \ldots \wedge brightness\_9\_9) \\ global\_rotation &\leftrightarrow (rotation\_0\_0 \wedge rotation\_0\_1 \wedge rotation\_0\_2 \wedge \ldots \wedge rotation\_9\_9) \\ global\_system &\leftrightarrow (global\_noise \vee global\_brightness \vee global\_rotation) \end{aligned}$$

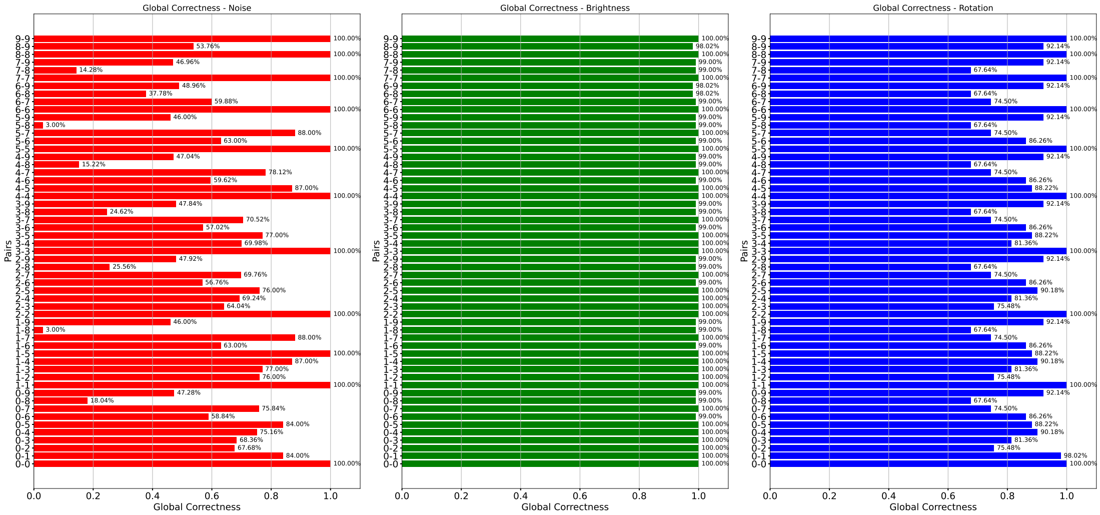

FIGURE 4.2: Global Correctness for Noise, Brightness, and Rotation Transformations

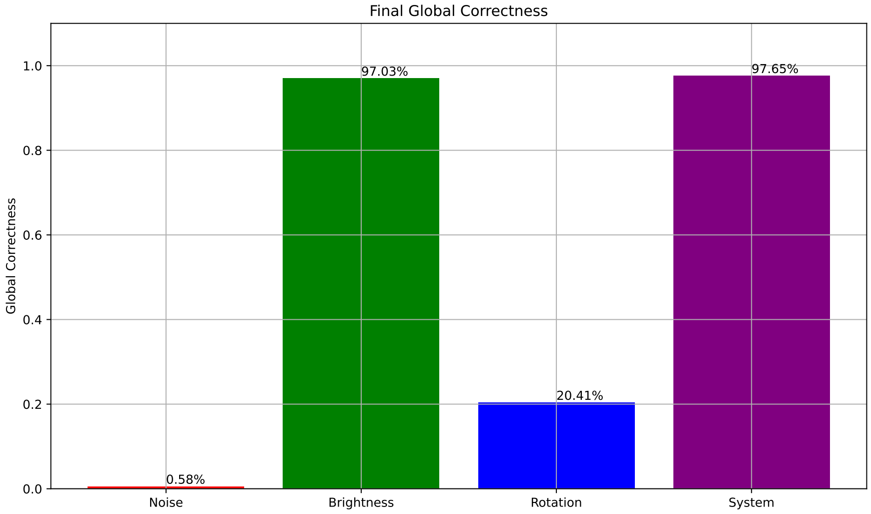

FIGURE 4.3: Final Global Correctness

The global correctness values for pairs of digits under the noise transformation vary significantly. Some pairs such as (0,0), (1,1), (2,2), etc., achieve perfect correctness (100%), while others like (0,5) and (1,8) exhibit very low correctness (3.00% and 18.04%, respectively). This indicates that the model's performance is highly inconsistent under noise, likely due to the distortion introduced affecting digit recognition. The model performs exceptionally well under brightness adjustments, with most pairs achieving near-perfect correctness. This suggests that the model is robust to changes in brightness, maintaining high confidence scores despite variations in image illumination. Performance under rotation is moderate, with significant variations across different pairs. Some pairs like (0,0), (1,1), (3,3), etc., achieve high correctness, while others such as (2,5) and (5,8) have lower correctness values. This variation suggests that while the model can handle certain rotations well, it struggles with others, potentially due to the angles and the inherent difficulty in recognizing rotated digits.

Combining the global correctness across all transformations, we observe an overall high performance with a final global correctness of 97.65%. **Brightness** contributes the most to this high score (97.03%), while Noise contributes the least (0.58%). Rotation has a moderate impact (20.41%), reflecting its varied performance across different pairs.

These observations highlight the strengths and weaknesses of the AI subsystem under different transformations. The high performance under brightness adjustments indicates robustness to illumination changes, while the challenges with noise and rotation suggest areas for improvement in handling these perturbations.

4.3 Use Case 2: Autonomous Vehicle Perception

In this use case, we investigate the robustness of an AI system for autonomous vehicles in detecting objects, particularly vehicles, under various weather conditions including fog, rain, snow, and sand. The goal is to assess the system's performance across these challenging environments to ensure reliable operation.

The dataset was split into training and validation sets, with class balancing performed through oversampling to address any class imbalances. Data augmentation techniques such as rotation, width and height shifts, shear transformation, zoom, and horizontal flips were applied to the training set to enhance the model's generalizability.

The trained model's performance was evaluated on a balanced validation set resampled to ensure uniform class distribution.

4.3.1 Local Correctness Analysis

The local correctness of the model for each weather condition is depicted in the first graph. The AI system exhibits the following correctness values: - **Fog:** The detection correctness is 51%, indicating moderate performance. Fog conditions likely introduce significant visual obstructions that challenge the model's ability to correctly identify vehicles. - **Rain:** Achieving a correctness of 75%, the model performs relatively well in rainy conditions, suggesting robustness to moderate visual disturbances caused by rain. - **Snow:** Similar to rain, the system shows a correctness of 75%, demonstrating its capability to handle snowy environments, where the visual contrast might be reduced. - **Sand:** With the highest correctness of 88%, the model excels in sandy conditions, possibly due to clearer visibility compared to other adverse weather scenarios.

4.3.2 Global Correctness Analysis

4.3.3 Global Correctness Evaluation for Vehicle Detection

The second graph illustrates the global correctness for vehicle detection under different combination methods of weather conditions:

$$global\_weather \leftrightarrow (fog \wedge rain \wedge snow \wedge sand)$$
$$global\_weather \leftrightarrow (fog \vee rain \vee snow \vee sand)$$
$$global\_weather \leftrightarrow (fog \wedge (rain \vee snow \vee sand))$$

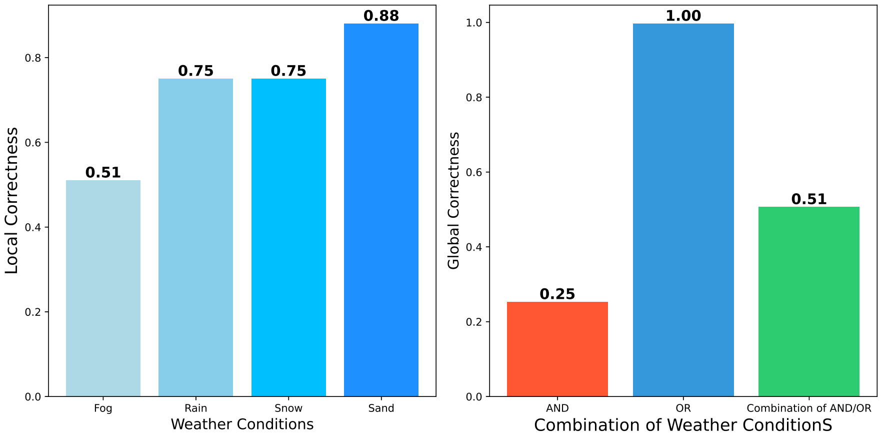

FIGURE 4.4: Local and Global Correctness of Vehicle Detection Under Various Weather Conditions

- **AND (Intersection):** The model's correctness is 25%. This low value reflects the stringent requirement for the system to correctly detect vehicles under all specified conditions simultaneously, highlighting potential vulnerabilities when multiple weather challenges are present.
- **OR (Union):** Achieving a perfect correctness of 100%, the model successfully detects vehicles under at least one of the conditions. This high performance underscores the system's reliability when at least one favorable condition is present.
- **Combination of AND/OR:** The correctness stands at 51%, representing a balanced approach where the model must correctly detect vehicles under fog and any one of the other conditions (rain, snow, or sand). This mixed strategy provides a realistic measure of the system's robustness in practical scenarios with varying weather conditions.

Chapter 5

References

Bibliography

[1] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y. and Alsaadi, F.E., 2017. A survey of deep neural network architectures and their applications. Neurocomputing, 234, pp.11-26.

[2] Hassija, V., Chamola, V., Mahapatra, A., Singal, A., Goel, D., Huang, K., Scardapane, S., Spinelli, I., Mahmud, M. and Hussain, A., 2024. Interpreting black-box models: a review on explainable artificial intelligence. Cognitive Computation, 16(1), pp.45-74.

[3] Liang, Y., Li, S., Yan, C., Li, M. and Jiang, C., 2021. Explaining the black-box model: A survey of local interpretation methods for deep neural networks. Neurocomputing, 419, pp.168-182.

[4] Ren, K., Zheng, T., Qin, Z. and Liu, X., 2020. Adversarial attacks and defenses in deep learning. Engineering, 6(3), pp.346-360.

[5] Tian, Y., Pei, K., Jana, S. and Ray, B., 2018, May. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In Proceedings of the 40th international conference on software engineering (pp. 303-314).

[6] Pei, K., Cao, Y., Yang, J. and Jana, S., 2017, October. Deepxplore: Automated whitebox testing of deep learning systems. In proceedings of the 26th Symposium on Operating Systems Principles (pp. 1-18).

[7] Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y. and Zhao, J., 2018, September. Deepgauge: Multi-granularity testing criteria for deep learning systems. In Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (pp. 120-131).

[8] Sun, Y., Huang, X., Kroening, D., Sharp, J., Hill, M. and Ashmore, R., 2018. Testing deep neural networks. arXiv preprint arXiv:1803.04792.

[9] Kim, J., Feldt, R. and Yoo, S., 2019, May. Guiding deep learning system testing using surprise adequacy. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp. 1039-1049). IEEE.

[10] Guo, J., Jiang, Y., Zhao, Y., Chen, Q. and Sun, J., 2018, October. Dlfuzz: Differential fuzzing testing of deep learning systems. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 739-743).

[11] Odena, A., Olsson, C., Andersen, D. and Goodfellow, I., 2019, May. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In International Conference on Machine Learning (pp. 4901-4911). PMLR.

[12] Sun, Y., Huang, X., Kroening, D., Sharp, J., Hill, M. and Ashmore, R., 2019, May. DeepConcolic: Testing and debugging deep neural networks. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 111-114). IEEE.

[13] Sekhon, Jasmine, and Cody Fleming. "Towards improved testing for deep learning." 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, 2019.

[14] Agarwal, Aniya, et al. "Automated test generation to detect individual discrimination in AI models." arXiv preprint arXiv:1809.03260 (2018).

[15] Sun, Youcheng, et al. "Concolic testing for deep neural networks." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018.

[16] Pei, Kexin, et al. "DeepXplore." Communications of the ACM 62.11 (2019): 137-145.

[17] Tian, Yuchi, et al. "Deeptest: Automated testing of deep-neural-network-driven autonomous cars." Proceedings of the 40th international conference on software engineering. 2018.

[18] Sun, Youcheng, et al. "Testing deep neural networks." arXiv preprint arXiv:1803.04792 (2018).

[19] Ma, Lei, et al. "Deepgauge: Multi-granularity testing criteria for deep learning systems." Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. 2018.

[20] Zhang, Mengshi, et al. "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018.

[21] Xie, Xiaofei, et al. "Deephunter: Hunting deep neural network defects via coverage-guided fuzzing." arXiv preprint arXiv:1809.01266 (2018).

[22] Gopinath, Divya, et al. "Symbolic execution for deep neural networks." arXiv preprint arXiv:1807.10439 (2018).

[23] Sato, T., Kameya, Y.: PRISM: A symbolic-statistical modeling language. In: IJCAI. pp. 1330–1339 (1997)

[24] Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)

[25] Vennekens, J., Denecker, M., Bruynooghe, M.: Logic programs with annotated disjunctions. In: ICLP. pp. 195–209 (2004)

[26] De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: IJCAI. pp. 2462–2467 (2007)

[27] Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence 64(1), 81–129 (1993)

[28] Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94(1-2), 7–56 (1997)

[29] Sato, T.: A statistical learning method for logic programs with distribution semantics. In: ICLP. pp. 217–232 (2001)

[30] Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming 11(2-3), 235–262 (2011)

[31] Vidal, G.: Explanations as Programs in Probabilistic Logic Programming. In: Hanus, M., Igarashi, A. (eds) Functional and Logic Programming. FLOPS 2022. Lecture Notes in Computer Science, vol 13215. Springer, Cham (2023)

[32] Arrieta, A.B., Rodríguez, N.D., Ser, J.D., et al.: Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. Information Fusion 58, 82–115 (2020)

Chapter 6

Current Progress

6.1 Achievements in the Last 10 Months

6.2 Challenges and Solutions

Chapter 7

Future Work and Two-Year Plan

7.1 Short-term Goals (Next 6 Months)

7.2 Medium-term Goals (Next 1 Year)

7.3 Long-term Goals (Next 2 Years)