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 Mni-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 learning (DL) models are critical in high-stakes 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 Bayesian network 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 DL 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.

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.

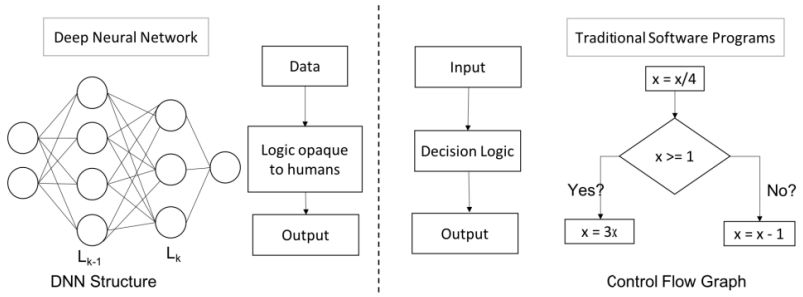

FIGURE 1.1: The internal logic of a deep neural network is opaque to humans, unlike the well-laid-out decision logic of traditional software programs [13]

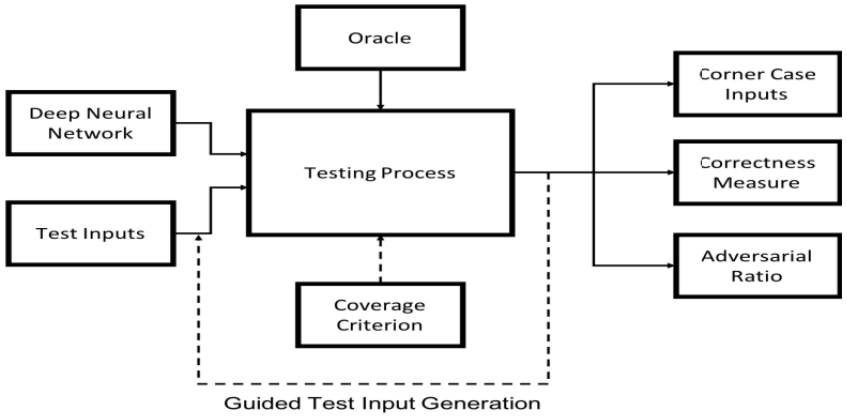

FIGURE 1.2: A high-level representation of most existing DNN testing methods [13]

- 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 9.

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.

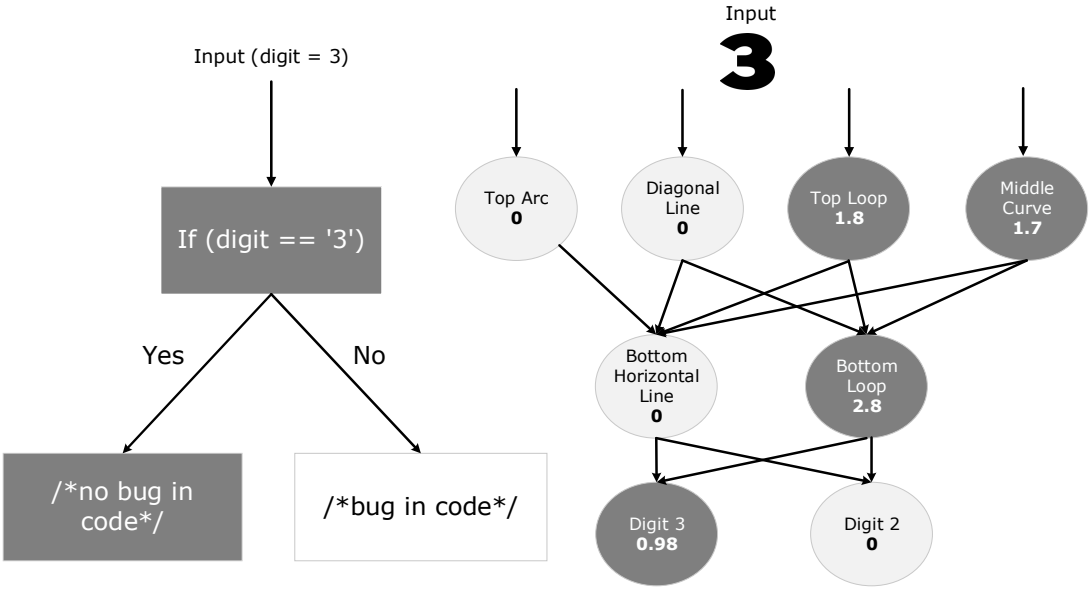

FIGURE 2.1: Comparison between program flows of a traditional program (left) and a neural network (right). The nodes in gray denote the corresponding basic blocks or neurons that participated while processing an input.

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 $\underset{i=1}{\overset{s_k}{\operatorname{argmax}}}\, y_i$, 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 $\mathcal{A}_{\text{adv}}$

(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.

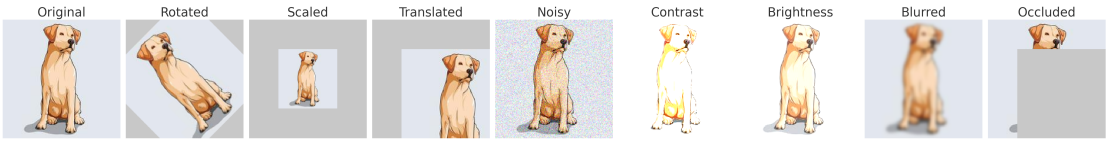

FIGURE 2.2: Image transformations for ...

- *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 Bayesian Networks

Bayesian networks are a powerful probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). They are particularly useful for modeling uncertainty in complex systems. The fundamental equation of Bayesian networks is Bayes' theorem, which is given by:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

This equation allows us to update the probability estimate for a hypothesis as more evidence or information becomes available.

In the context of DNNs, Bayesian networks can offer significant advantages. Behavioral assessment considers probabilistic relationships to offer a richer, more detailed understanding of how different inputs affect model outputs. By developing a strategic focus on high-risk areas, it prioritizes regions with the highest uncertainty or risk for targeted testing, making the process more efficient and effective.

2.1.5 Probabilistic Logic Programming and Problog

Probabilistic logic programming (PLP) combines logic programming with probability theory, enabling the representation of uncertain knowledge. Problog, a

prominent PLP language, extends Prolog by associating probabilities with facts and rules, facilitating the handling of uncertain information effectively.

Problog was introduced by De Raedt et al. [23], providing a framework for probabilistic reasoning within the well-established logical framework of Prolog. The key contributions of Problog include:

- **Integration of Probabilities:** Problog integrates probabilistic reasoning with logic programming by allowing facts to have associated probabilities.
- **Flexible Inference Mechanism:** It supports various inference techniques, including exact and approximate inference, to handle different types of queries and data efficiently.
- **Expressiveness:** Problog's expressiveness allows it to model complex relationships and dependencies, making it suitable for a wide range of applications.

2.1.6 Prolog to Python Syntax Conversion

Clauses in Problog can be constructed by overloading Python operators, making the syntax more intuitive for those familiar with Python. The following table illustrates the conversion:

Prolog	Python	English
:-	<<	clause
,	&	and
;	|	or
\+	~	not

TABLE 2.1: Prolog to Python Syntax Conversion

2.1.7 Example Scenario

In this section, we illustrate how Problog can be utilized to find global robustness based on local robustness values obtained from a deep learning model.

Consider a scenario where we have the local robustness values for two digits, digit 0 and digit 1, obtained using a deep learning model in Python. We use Problog to find the global robustness based on these local values.

Here is an example Problog code snippet:

```
0.8::digit_0.
0.6::digit_1.
```

```
correct_0 :- digit_0.
wrong_0 :- \+correct_0.
correct_1 :- digit_1.
wrong_1 :- \+correct_1.

pair_0_1 :- correct_0, correct_1.
pair_wrong_0_1 :- wrong_0, wrong_1.

global_correct :- pair_0_1; pair_wrong_0_1.

query(correct_0).
query(correct_1).
query(wrong_0).
query(wrong_1).
query(pair_0_1).
query(pair_wrong_0_1).
query(global_correct).
```

In this example:

- `0.8::digit_0.` specifies that digit 0 has a local robustness probability of 0.8.
- `0.6::digit_1.` specifies that digit 1 has a local robustness probability of 0.6.
- `correct_0` and `correct_1` indicate that the predictions for digits 0 and 1 are correct.
- `wrong_0` and `wrong_1` indicate that the predictions for digits 0 and 1 are wrong.
- `pair_0_1` checks if both digits are correctly predicted.
- `pair_wrong_0_1` checks if both digits are wrongly predicted.
- `global_correct` determines if the global prediction is correct by checking if both digits are either correctly or wrongly predicted.
- The `query` statements are used to query the probabilities of these events.

Using Problog in this way allows us to compute the global robustness of the system based on the local robustness of individual components, providing a more comprehensive assessment of the model's performance under uncertainty.

Chapter 3

Proposed Framework

3.1 Proposed Approach

FIGURE 3.1: Overview of the Proposed Framework

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.

Example 3.1. *An instance of a simple AI system is an* MNIST Digit Adder $\mathcal{S}_{MNIST} = (\mathcal{F}, \mathcal{D})$*, where* $\mathcal{F} = (\{f_{mnist}\}, +)$*,* 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.,* $\mathcal{F}(x_1, x_2) = f_{mnist}(x_1) + f_{mnist}(x_2)$*.* $\mathcal{D}$ *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* $\mathcal{S}_{MNIST}$*, 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 $\mathcal{S}_{MNIST}$ 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 $f_{mnist}(x) = c$. 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 $\mathcal{S}_{MNIST}$, 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 $x_1' = rotate(x_1, \theta)$ and $x_2' = rotate(x_2, \theta)$. The test case is then: $\mathcal{T}_{rotation} = \{(x_1', x_2') \mid x_1', x_2' \in rotate(S, \theta)\}$*
- ***Noise**: For a given mean μ and standard deviation σ, generate the perturbed images $x_1'' = noise(x_1, \mu, \sigma)$ and $x_2'' = noise(x_2, \mu, \sigma)$. The test case is then: $\mathcal{T}_{noise} = \{(x_1'', x_2'') \mid x_1'', x_2'' \in noise(S, \mu, \sigma)\}$*

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

3.4 Validation

In this step, we run the validation queries for each test case generated previously. Let f_i be a classifier, x an input, and c the correct class label of x. Denote by $query(f_i, x, c)$ the outcome of the validation query defined as 1 if $f_i(x) = c$ and as 0 otherwise.

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

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

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

3.4.0.1 Local Correctness

Local correctness involves checking the AI subsystem's performance on individual images subjected to different transformations. For each image x from a set of samples X, the AI subsystem produces a confidence score $f(x)$ representing its certainty in recognizing the digit. The local correctness for a transformation T applied to an image x is defined as:

$$\text{Local Correctness}(x, T) = f(T(x))$$

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.

3.4.0.2 Global Correctness

Global correctness evaluates the AI subsystem's performance on combinations of images and transformations, such as adding two digits under different transformations. Given a set of 100 samples for each digit (0-9), we randomly select images to form digit pairs. The global correctness assesses the combined confidence for recognizing the result of operations like addition under each transformation.

For each transformation T, we define local correctness for digits d_1 and d_2 as follows:

$$\mathrm{LC}_T(d_1) = \text{Local Correctness}(x_1, T)$$

$$\mathrm{LC}_T(d_2) = \text{Local Correctness}(x_2, T)$$

where x_1 and x_2 are randomly selected images representing digits d_1 and d_2.

The global correctness for each transformation is then defined as the product of the local correctness values:

$$\mathrm{GC}_T = \mathrm{LC}_T(d_1) \times \mathrm{LC}_T(d_2)$$

To determine the optimistic and pessimistic global correctness across all transformations, we calculate the following:

- **Optimistic Global Correctness**:

$$\mathrm{GC}_{opt} = \max_T(\mathrm{GC}_T)$$

- **Pessimistic Global Correctness**:

$$\mathrm{GC}_{pes} = \min_T(\mathrm{GC}_T)$$

For the addition operation, if we denote the sum of digits d_1 and d_2 as d_{sum}, the overall global correctness for the operation is:

$$\text{Overall Global Correctness} = \min(\mathrm{GC}_{opt}, \mathrm{GC}_{pes})$$

where each transformation is evaluated separately, and the global correctness is determined based on the individual probabilities of correct recognition for each digit under each transformation.

3.5 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).

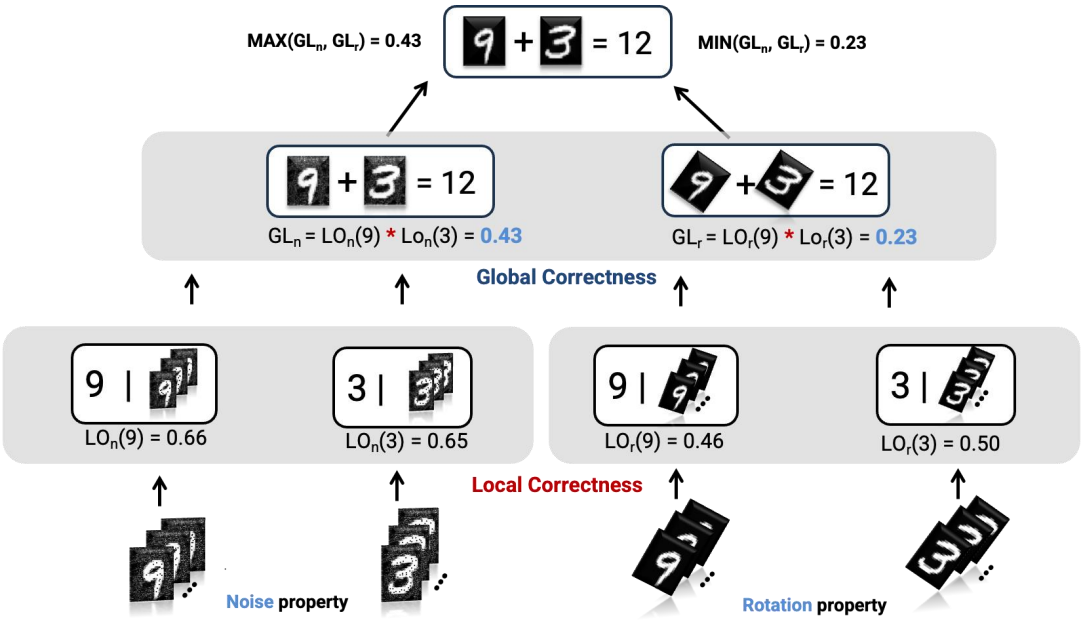

FIGURE 3.2: Graphical View of Local and Global Correctness

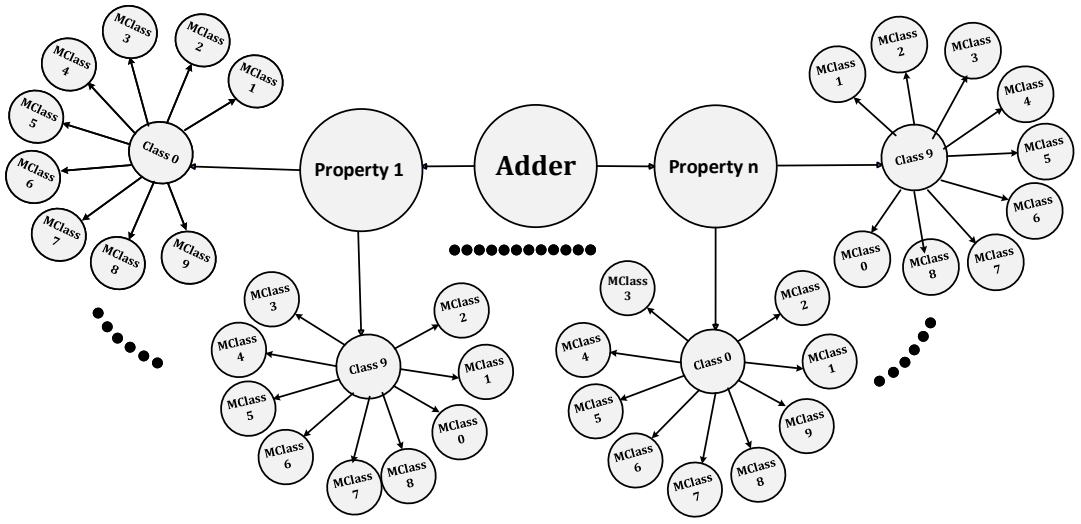

FIGURE 3.3: Error Summarization

3.6 Gradient-Based Test Generation

Algorithm 1: Test Case Generation via Gradient-Based Attacks

Input: Model $\mathcal{M}$ with bounds [0, 1],
Set of images $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$,
Corresponding labels $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$,
Perturbation magnitudes $\mathcal{E} = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_k\}$,
Set of attacks $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$.

Output: Set of test cases $\bigcup TestCases_{ij}$.

Procedure: GenerateTestCases($\mathcal{M}$, $\mathcal{I}$, $\mathcal{L}$, $\mathcal{E}$, $\mathcal{A}$)

```
for each attack A_j in A
    for each ε_i in E
        Generate testcases Adv_ij = A_j(M, I, ε_i)
        Verify the Adv_ij to obtain V_ij
        Evaluate V_ij against L to determine isRobust_ij
        Compile test cases TestCases_ij = {Adv_ij, isRobust_ij}
    end for
end for
return ⋃ TestCases_ij
```

Chapter 4

Simulations and Results

Chapter 5

Conclusion

5.1 Summary of Current Research

5.2 Impact and Contributions

5.3 Final Thoughts

Chapter 6

References

References

[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] De Raedt, L., Kimmig, A. and Toivonen, H., 2007, January. Problog: A probabilistic prolog and its application in link discovery. In IJCAI (Vol. 7, pp. 2462-2467).

[24] Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T. and De Raedt, L., 2018. Deepproblog: Neural probabilistic logic programming. Advances in neural information processing systems, 31.

Chapter 7

Current Progress

7.1 Achievements in the Last 10 Months

7.2 Challenges and Solutions

Chapter 8

Future Work and Two-Year Plan

8.1 Short-term Goals (Next 6 Months)

8.2 Medium-term Goals (Next 1 Year)

8.3 Long-term Goals (Next 2 Years)

Chapter 9

Conclusion