Journal of Information Science and Engineering XX, XXXX-XXXX (2024)
DOI: xx.xxx.xxxx.xx

Concolic Testing on Individual Fairness of Neural Network Models

Ming-I Huang, Chih-Duo Hong, and Fang Yu
National ChengChi University
Taipei, Taiwan
E-mail: {111356047; chihduo; yuf}@nccu.edu.tw

This paper introduces PyFair, a formal framework for evaluating and verifying individual fairness of Deep Neural Networks (DNNs). By adapting the concolic testing tool PyCT, we generate fairness-specific path constraints to systematically explore DNN behaviors. Our key innovation is a dual network architecture that enables comprehensive fairness assessments and provides completeness guarantees for certain network types. We evaluate PyFair on 25 benchmark models, including those enhanced by existing bias mitigation techniques. Results demonstrate PyFair's efficacy in detecting discriminatory instances and verifying fairness, while also revealing scalability challenges for complex models. This work advances algorithmic fairness in critical domains by offering a rigorous, systematic method for fairness testing and verification of pre-trained DNNs.

1. Introduction

Deep Neural Networks (DNNs) are increasingly deployed across diverse applications, from autonomous vehicles to medical diagnostics. While these models often achieve remarkable performance, their deployment in high-stakes scenarios raises significant concerns about trustworthiness and fairness. In critical domains such as criminal justice, employment, and financial services, the algorithmic fairness of DNNs has come under intense scrutiny [1, 2, 3]. Notable examples include racial bias observed in the COMPAS recidivism prediction model [4] and gender bias evident in Amazon's recruiting model [5]. This heightened awareness underscores the critical importance of addressing fairness issues of neural networks.

Extensive efforts have been directed towards enhancing individual fairness at the model level. One prevalent approach is fairness testing, which aims to generate efficient test suites before deployment [6, 7, 8, 9, 10]. These discrimination tests can be employed to quantify discrimination or can be utilized for model retraining to alleviate unfairness. However, such methods face challenges due to substantial time overheads, thereby compromising the efficiency of the remediation process. Tools like Faire [11] aim to capture the cause of unfairness and use condition checks to rectify individual discrimination. Despite these efforts, verifying fairness properties in complex DNNs remains challenging due to their non-linear nature and the extensive scope of fairness requirements.

Various forms of discrimination are acknowledged, encompassing group discrimination [12] and individual discrimination [13]. Discrimination is typically delineated with respect to a set of *Protected Attributes (PAs)*, such as race and gender, in contrast to the

Non-Protected Attributes (NPAs). Group fairness advocates for impartial treatment among protected groups (i.e., the sub-population defined by a protected attribute) to eliminate group discrimination. While group fairness is relatively easy to measure and has clear policy implications, it can sometimes mask individual-level disparities or lead to reverse discrimination. In contrast to group fairness, individual fairness dictates that similar individuals should be treated similarly regardless of their membership in protected groups. In the context of machine learning, this means that two inputs differing solely in their PAs should lead to identical model outcomes.

This paper presents PyFair, a novel framework for evaluating individual fairness of DNNs using the concolic testing tool PyCT [14, 15]. Our research addresses the critical gap in formal fairness guarantees for real-world DNN applications by extending automatic testing techniques, previously successful in safety property assurance, to fairness verification. Our methodology leverages concolic testing [16], which combines concrete execution with symbolic analysis to explore DNN behaviors systematically. The key innovation lies in adapting concolic execution to generate fairness-specific path constraints. By systematically exploring these constraints, we identify critical test inputs traversing diverse DNN decision paths, enhancing coverage of inputs most relevant to fairness assessment.

Even though PyCT can certify the individual fairness of all given test samples, the absence of detected discrimination generally does not guarantee overall model fairness. The reason is that PyCT can only perturb one PA per sample while keeping NPAs fixed. Hence, it only explores test cases whose NPA values coincide with the given sample. To address this limitation, we introduce the PyFair concolic testing framework, which utilizes a dual network architecture to examine any feasible discriminatory instance with identical NPAs but different PAs. PyFair extends the capabilities of PyCT, enabling a more thorough and systematic exploration of the model's behavior beyond random sampling. Thanks to our employment of SMT (Satisfiability Modulo Theories) solvers [17], this exploration is complete when the neural network can be faithfully encoded in an SMT theory.[1]

We assess the efficacy of PyFair by evaluating network models studied in the literature [18, 8, 9]. Our experiments demonstrate that PyFair can effectively identify discriminatory instances in most models, often outperforming existing constraint-based testing tools like Fairify [18]. We also test models improved by bias mitigation techniques such as ADF [8] and EIDIG [9], showing that PyFair can still detect unfairness in these "fairer" models. Furthermore, we evaluate PyFair's capability to verify fairness in artificially constructed fair models, revealing both its potential and limitations in handling complex architectures. These comprehensive experiments showcase PyFair's effectiveness in both discriminatory instance detection and fairness verification.

Overall, this work advances the state-of-the-art in neural network fairness verification through a rigorous, systematic approach to identifying discriminatory instances. Our framework's dual network architecture, combined with SMT solvers, enables comprehensive exploration of fairness properties with theoretical guarantees. However, while our approach delivers stronger guarantees than random sampling or gradient-based methods, it requires greater computational resources and faces scalability challenges when dealing with complex network architectures. This tradeoff between rigorous verification and

[1] For example, DNNs that use ReLU as its activation functions and Softmax in its output layer can be faithfully encoded in the theory of linear real arithmetic for classification tasks.

Table 1: Summary of related work in fairness testing and verification

Approach	**Focus**	**Methodology**
THEMIS [6]	Fairness testing	Causality-based random sampling
AEQUITAS [7]	Fairness testing	Local and global random sampling
SymbGen [19]	Fairness testing	Test case generation using symbolic execution
ADF [8]	Bias mitigation	Adversarial sampling based on gradient search
EIDIG [9]	Bias mitigation	An ADF variant with momentum optimization
NeuronFair [10]	Fairness testing	Adversarial sampling via neuron interpretation
Fairify [18]	Fairness verification	Adversarial sampling based on constraint-solving
PyFair	Testing & verification	Adversarial sampling based on concolic testing

computational efficiency represents an important consideration for practitioners choosing between different fairness testing approaches based on their specific requirements for completeness versus scalability.

2. Related Work

Fairness testing. Recent research has focused extensively on testing and validating the fairness of DNNs using discriminatory examples. THEMIS [6] introduces fairness scores as measurement metrics of fairness and devises a causality-based algorithm for random discriminatory sample generation. While THEMIS uses pure and unguided random sampling, tools like AEQUITAS [7] and SymbGen [19] offer more targeted generation algorithms to identify fairness violations. AEQUITAS pioneers a two-step approach combining global and local search strategies, while SymGen exploits symbolic execution and local explanability to generate effective test cases. Adversarial sampling [20] is also a popular method for analyzing fairness of DNNs. ADF [8] adopts a two-phase gradient search to identify discriminatory examples. EIDIG [9] furthermore optimizes ADF by incorporating momentum in the global generation phase and reducing the frequency of gradient calculations in the local generation phase. Despite these advancements, ADF and EIDIG suffer from the issues of gradient vanishing and local optima. To address these challenges, NeuronFair [10] interprets internal DNN states to guide instance generation and explore decision boundaries. Unlike black-box methods such as THEMIS and AEQUITAS, which prioritize efficiency through random sampling, PyFair provides more insights via white-box analysis at a higher computational cost. Compared to heuristic search methods like ADF and EIDIG, PyFair provides formal completeness guarantees, offering a more systematic exploration of discriminatory instances using concolic testing and SMT solvers.

Fairness verification. Testing has proven helpful in identifying fairness violations and addressing model deficiencies, but it often falls short of verifying the absence of fairness violations. Most studies in fairness verification have focused on group fairness, as seen in FairSquare [21] and VeriFair [22]. John et al. [23] present the first technique for verifying individual fairness of classical machine learning models. For neural networks, LCIFR [24] certifies individual fairness by formulating it as a local property, which coincides with robustness within a specific distance metric. In contrast, Libra [25] computes certifications for the global property of causal fairness. Conceptually, ensuring individual fairness entails verifying a local or global robustness property, wherein the classifier output remains unchanged for perturbations of any input within the domain. Fairify [18]

Algorithm 1: Discriminatory Instance Checking

```
Input: DNN Model $M$, Dataset $\Phi$, Protected Attributes $PA$
Output: An unfairness witness $(\varphi, \varphi')$, if any
foreach individual $\varphi$ in $\Phi$ do
    Initialize $Q, T$ ;                                   // Both are empty at the beginning
    $Q, T \leftarrow$ Exploration($M, \varphi, PA, Q, T$); ;   // Explore paths induced by $\varphi$ (see [15])
    while $Q$ is not empty do
        $\phi \leftarrow Q$.dequeue() ;                   // $\phi$ is the next path constraint
        if $\phi$ has a solution then
            $\varphi' \leftarrow$ result from SMT Solver ;  // $\varphi'$ enables a new path
            if $M(\varphi) \neq M(\varphi')$ then
                Abort and report $(\varphi, \varphi')$ ;    // An unfairness witness is found
            else
                $Q, T \leftarrow$ Exploration($M, \varphi', PA, Q, T$) ; // Explore paths induced by $\varphi'$
Abort and report no discriminatory instances in Dataset $\Phi$ ;   // No unfairness found in $\Phi$
```

is a constraint-based approach for verifying the individual fairness of DNNs. The tool decomposes the verification task into multiple sub-problems and prunes the networks to mitigate verification complexity. While designed primarily for testing, our method can also be employed to certify model fairness by concluding that there are no discriminatory instances for a model. Another line of research attempts to achieve individually fair models through enforcement during model training [26, 24, 11, 27, 28]. Although this work focuses primarily on determining whether a pre-trained DNN model violates fairness, our method can be easily integrated into existing fair training approaches. Indeed, discriminatory instances identified by our method can be used in model training and refinement, such as generating challenging test cases to evaluate the trained models and identifying areas where fairness enforcement might fall short.

Concolic testing. Concolic testing has been adapted for neural networks to explore execution paths and increase test coverage [29]. Tools such as DeepXplore [16], DeepGauge [30], and DeepCon [31] offer alternative avenues by generating adversarial examples that expose vulnerabilities in neural networks. DeepXplore introduces neuron coverage as a metric for measuring DNN testing adequacy and uses multiple similar DNNs as cross-referencing oracles to avoid manual checking. DeepCon proposes contribution coverage, which considers both neuron outputs and connection weights to gauge testing adequacy. DeepConcolic [32] conducts symbolic execution testing based on neuron coverage, generating inputs that activate neurons not triggered in the current execution. By combining gradient-based and constraint-based methods, DeepConcolic systematically maximizes neuron coverage across various paths. Most existing DNN testing tools focus on maximizing certain coverage measures instead of exploring critical branches for changing prediction outcomes. As a result, it is not straightforward to plug these tools into our framework to detect discrimination concerning protected attributes.

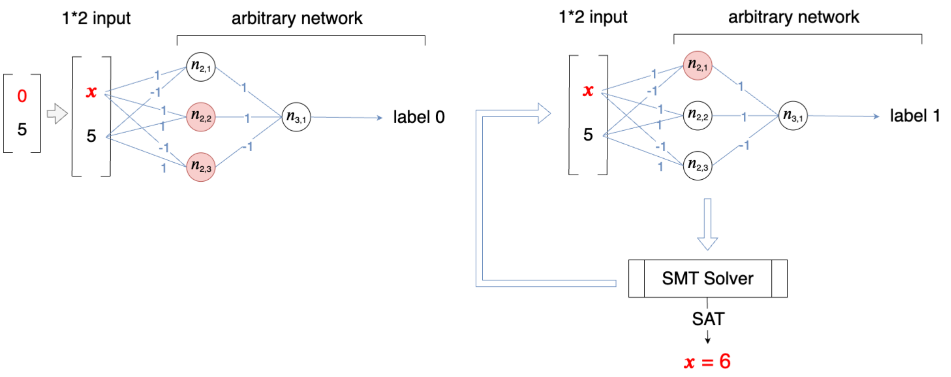

Fig. 1: A 3-layer DNN where the first attribute is protected (left). For this DNN and an input φ: [0, 5], PyCT identifies another input φ': [6, 5] with a different model output (right), which indicates that φ is a discriminatory instance for the DNN.

3. Discriminatory Instance Checking with PyCT

In this section, we detail our methodology for evaluating individual fairness in DNNs through discriminatory instance checking. Our objective is to identify discriminatory instances for a pre-trained model, serving as evidence of the model's unfairness. We denote the attributes of the model input as $A = \{A_1, A_2, ..., A_n\}$. Each attribute A_i is associated with a domain I_i. The input domain is $I = I_1 \times I_2 \times \cdots \times I_n$, representing all possible combinations of the attribute values. We use $PA \subset A$ to represent the set of Protected Attributes (PAs) such as gender, race, and age. We use $NPA = A \setminus PA$ to denote the set of Non-Protected Attributes (NPAs).

Definition 3.1 (Discriminatory Instance [8, 11])**.** Let $\varphi = (a_1, a_2, ..., a_n)$ denote an arbitrary instance in the dataset, where a_i represents the value of attribute A_i. Given a model M, we say φ is a *discriminatory instance* of M if there exists $\varphi' = (a'_1, a'_2, ..., a'_n)$ such that (i) φ, φ' belong to the input domain of M, (ii) $\exists A_i \in PA.\ a_i \neq a'_i$, (iii) $\forall A_i \in NPA.\ a_i = a'_i$, and (iv) $M(\varphi) \neq M(\varphi')$. We say (φ, φ') is an *unfairness witness* of M. A model is *unfair* if it has an unfairness witness.

A *Deep Neural Network (DNN)* consists of an input layer, multiple hidden layers, and an output layer. Neurons in each layer connect to those in the adjacent layer through weighted connections, enabling information extraction and transformation. Frequently used activation functions include Rectified Linear Unit (ReLU), Sigmoid, Hyperbolic Tangent (Tanh), and Softmax. The increased depth and complexity of DNNs make them particularly effective for advanced tasks such as image processing, computer vision, and natural language processing. PyCT is engineered to parse and simulate the operations of network models embedded within Python programs, which includes implementing commonly used activation functions such as ReLU and Sigmoid. As a result, PyCT can analyze neural network models and Python programs in an integrated and coherent manner.

Figure 1 illustrates how PyCT checks if a given input is discriminatory for a DNN.

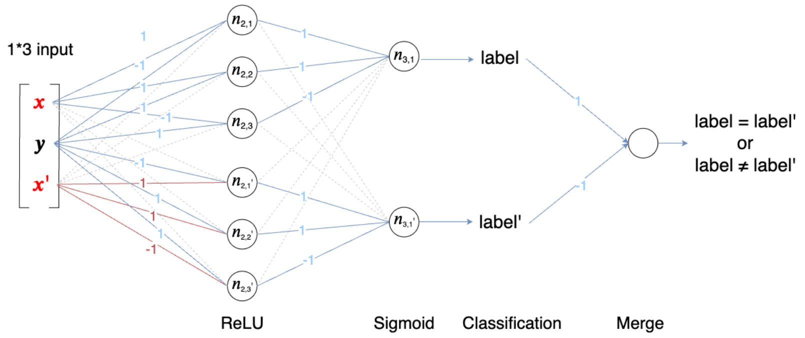

Fig. 2: The 2-DNN obtained from the DNN in Figure 1

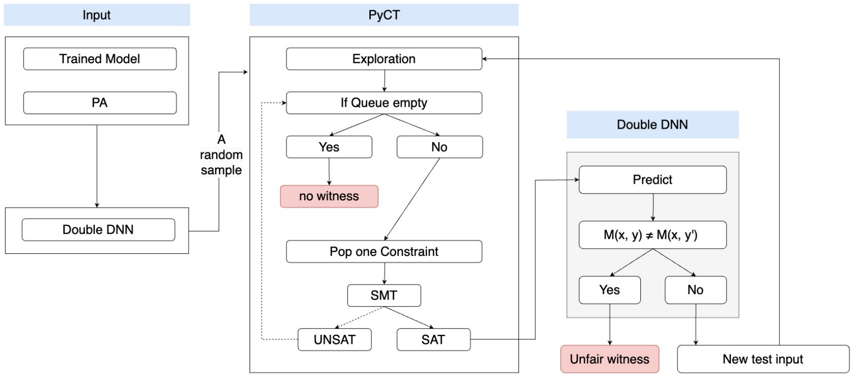

Fig. 3: An overview of the PyFair framework

The example depicts a DNN with a 1×2 input, undergoing a ReLU operation followed by a Sigmoid function for binary classification. A sample $[0, 5]$ is given for discrimination evaluation, with the first attribute designated as protected (marked in red). We make the protected attribute value 0 a concolic variable $(0, x)$, which allows the perturbation of x to identify attribute values that might alter the model's output. The core algorithm for testing discriminatory instances is presented in Algorithm 1. Essentially, PyCT maintains a tree T to track the path constraints associated with all explored network paths. It also manages a queue Q containing formulas whose solutions correspond to input values that guarantee coverage of previously unexplored network paths. PyCT employs an SMT solver [33, 17] to solve these formulas and find new test inputs that satisfy previously unexplored branch conditions. In this example, the solver identifies a solution $x = 6$ for the perturbed input, leading to a new test case $[6, 5]$. Feeding this case into the model changes its output from 0 to 1. Consequently, φ is a discriminatory instance and we conclude that the DNN is unfair.

4. Fairness Verification with PyFair

Model fairness checking aims to exhaustively explore the input domain to identify a pair of instances φ and φ' identical in their NPAs but differ in their PAs and model outcomes. As discussed in the previous section, PyCT can be directly applied to check model fairness. However, since PyCT can only check one instance at a time, it faces limitations when dealing with infinite input domains: even after testing numerous samples without detecting discrimination, PyCT often cannot conclusively establish model fairness. To address this limitation, we propose a framework, named PyFair, to certify model fairness with completeness guarantees. This framework essentially extends PyCT with an innovative data structure called the Dual DNN.

4.1 The Dual-DNN Architecture

A *Dual DNN (2-DNN)* is a DNN with three types of attributes PA, NPA, and PA', where PA' duplicates PA. A 2-DNN $\tilde{M}$ is constructed by creating two copies of a given DNN M, such that one takes an input on PA and NPA, and the other takes an input on PA' and NPA. The outputs of these two DNNs are combined by comparison to produce the final output of $\tilde{M}$. As an illustration, consider the 2-DNN $\tilde{M}$ in Figure 2, which is derived from the DNN M in Figure 1. Given a (symbolic) input instance (x, y) with $x \in$ PA and $y \in$ NPA, the 2-DNN transforms it to (x, y, x'), passes x, y to the nodes $n_{2,1}, n_{2,2}, n_{2,3}, n_{3,1}$, and passes x', y to the nodes $n'_{2,1}, n'_{2,2}, n'_{2,3}, n'_{3,1}$. The 2-DNN's output is determined by

$$\tilde{M}(x, y, x') = \begin{cases} 1, & M(x, y) \neq M(x', y) \\ 0, & o.w. \end{cases} \tag{1}$$

Subsequently, we can employ PyCT to explore the network $\tilde{M}$ and identify a solution for $\tilde{M}(x, y, x') = 1$ with $x \neq x'$. If all branches result in UNSAT, we conclude that the original model M has no unfairness witness. In this way, we can transform the fairness checking problem of a DNN into a problem of finding adversarial examples for a 2-DNN.

Algorithm 2 describes how to construct a 2-DNN from a DNN and a set of PA indices. The primary task is to compute the network parameters (i.e., the weight matrix), as the rest of the steps are straightforward. The approach merges two replicas of M, sharing a common input, while setting zero for connections that should not transmit values to prevent interference. The weight adjustments apply across input, hidden, and bias layers. In the input layer, the row count remains unchanged to avoid duplicating features, accommodating additional PAs. A matrix with dimensions [input size × 2·(first hidden layer size)] is initialized (line 7). Each row of weights is duplicated (lines 8-10), while original PA weights bypass new hidden nodes (line 18). New rows transmit new PA weights to added nodes only (lines 19-21), enabling dual inputs where NPA values remain constant but PA values differ. Hidden layers are expanded by doubling row and column sizes (lines 4 and 6). Weights in the first half of each row are duplicated with zeros in the second half (line 9), while columns follow an inverse pattern (line 10). This configuration ensures the hidden layers are appropriately adapted for 2-DNN operations. For bias layers, each node's bias is simply duplicated (lines 24-25) to align with the replicated structure of the 2-DNN.

Algorithm 2: Computation of 2-DNN

```
Input: Original model $M$ and a list of PA indices $PA$
Output: The weight matrix $w_2$ of the 2-DNN for $M$
$w \leftarrow$ the weight matrix of $M$ ;                  // Retrieve weights of the original model
Initialize an empty list $w_2$ ;                           // List to store 2-DNN weights
foreach layer index $l$ in $w$ do
    $m, n \leftarrow$ size($w[l]$), size($w[l+1]$) $\times$ 2 ;   // Determine dimensions
    if $l$ is the index of a hidden layer then
        $m \leftarrow m \times 2$ ;                        // Double the rows for the hidden layer
        $z \leftarrow$ a new array of shape $(m, n)$ ;      // Create array to store weights
        for $i \leftarrow 0$ to $m - 1$ do
            $z[i][: n] \leftarrow w[l][i]$ ;               // Copy original weights to the first half
            $z[i][n :] \leftarrow w[l][i]$ ;               // Duplicate weights to the second half
        Append $z$ to $w_2$ ;                              // Store the updated weights
    else
        if $l$ is the index of an input layer then
            $z \leftarrow$ a new array of shape $(m, n)$ ;  // Create array for input layer weights
            for $i \leftarrow 0$ to $m - 1$ do
                $z[i][: n] \leftarrow w[l][i]$ ;           // Copy original weights to the first half
                $z[i][n :] \leftarrow w[l][i]$ ;           // Duplicate weights to the second half
            for $p$ in $PA$ do
                $z[p][n :] \leftarrow 0$ ;                 // Set protected attributes' second half to zero
                $a \leftarrow [0] * n + w[l][p]$ ; // Create additional row for protected attributes
                Append list $a$ to the end of $z$ ;        // Add new row for protected attributes
            Append $z$ to $w_2$ ;                          // Store the updated input layer weights
        else
            $z \leftarrow$ a copy of $w[l]$ ;              // $l$ is a bias layer; simply copy weights
            Append $z$ to $w[l]$ ;                         // Duplicate the layer weights
            Append $w[l]$ to $w_2$ ;                       // Store the duplicated weights
return $w_2$ ;                                             // Return the constructed 2-DNN weights
```

4.2 The PyFair Framework

We outline the framework of PyFair in Figure 3. PyFair takes a DNN model and a set of PAs as input. It first constructs a 2-DNN based on the model and the PAs. Starting from a random concolic input, PyFair either finds an unfairness witness or reports that no such witness exists. This approach is complete for checking model fairness when the input DNN can be faithfully encoded in an SMT theory (e.g., when the network has ReLU as its activation functions and Softmax in its output layer).

Proving unfairness with PyFair. As a detailed example, we describe the execution of PyFair on the 2-DNN in Figure 2 and the input $[0, 5, 0]$. As mentioned earlier, the concolic tester maintains a tree T and a queue Q to explore previously unexamined network branches. Figure 4 depicts the states of T and Q during discriminatory instance checking.

The 1^{st} *iteration:* We set three input as concolic variable: $(0, x)$, $(5, y)$, and $(0, x')$. After computing the weighted sum, the first hidden node $n_{2,1}$ remains a concolic variable

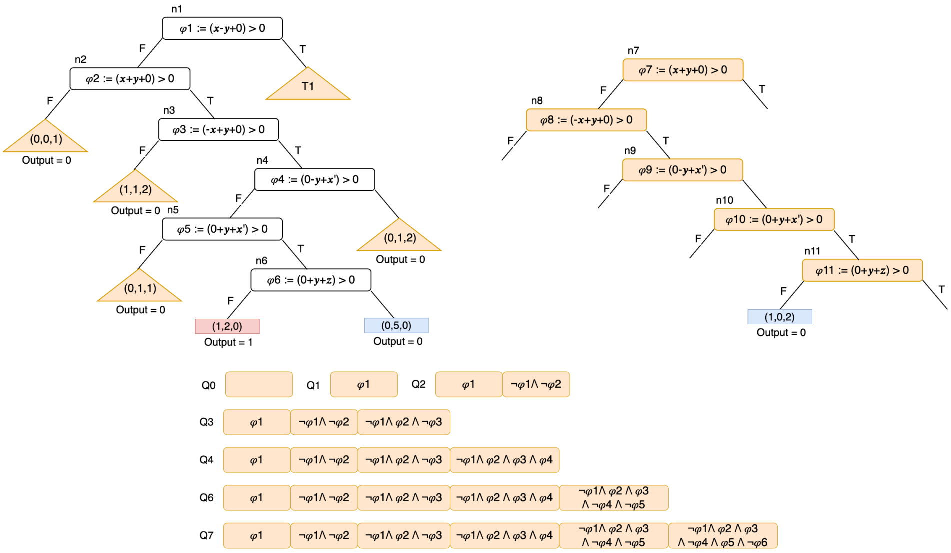

Fig. 4: Tree T (top left), Tree T' (top right), and Queue Q (below)

$(-5, x-y)$, computed as $0*1+5*(-1)+0*0 = -5$ for the concrete value and $x*1+y*(-1)+x'*0 = x-y$ for the symbolic expression. A node $n_{2,1}$ with label $\varphi_1 = (x-y) > 0$ is inserted into T as a root. The remaining hidden nodes $n_{2,1}, n_{2,2}, n_{2,3}, n_{3,1}, n'_{2,1}, n'_{2,2}, n'_{2,3}, n'_{3,1}$ are computed in similar manner. The output vector is computed using the absolute difference between the variables $label$ and $label'$ to determine if the two labels are the same. Since our input is $(0,5,0)$, we collect the input along with its output as $x = 0$, $y = 5$, $x' = 0$, and $label = label'$.

The 2^{nd} *iteration:* We give the constraints dequeued from Q to the SMT solver, which provides a solution $x = 1, y = 0, x' = 2$ as a new test input. We repeat this procedure to gather more branches, as illustrated by Tree T' in Figure 4. Finally, the Sigmoid function produces the same prediction as the original label. Thus, we continue dequeuing constraints from Q.

The $3^{rd} \sim 6^{th}$ *iterations:* Constraints $\neg\varphi_1 \wedge \neg\varphi_2$, $\neg\varphi_1 \wedge \varphi_2 \wedge \neg\varphi_3$, etc., are dequeued from Q in order by exploring new test inputs. This process terminates when the queue Q is empty, or the output label is 1 (indicating the discovery of a unfairness witness). Since the outputs remain unchanged for these inputs, the process continues.

The 7^{th} *iteration:* In this iteration, the constraint $\neg\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \neg\varphi_4 \wedge \varphi_5 \wedge \neg\varphi_6$ is dequeued from Q. This time, the test input $x = 1, y = 2, x' = 0$ yields a different model prediction. We add a node with this input and the prediction label $Output = 1$ as the left child of n6. The process terminates here and reports unfairness for the model. The witness of unfairness is $((1,2),(0,2))$.

Proving fairness with PyFair. We proceed to demonstrate how PyFair certifies the fairness of a fair model. To obtain a fair model, we simply modify the DNN in Figure 1 by setting the outgoing weights of the PA to zero. With this setup, we can guarantee that the PA value does not influence the output value, resulting in a fair model.

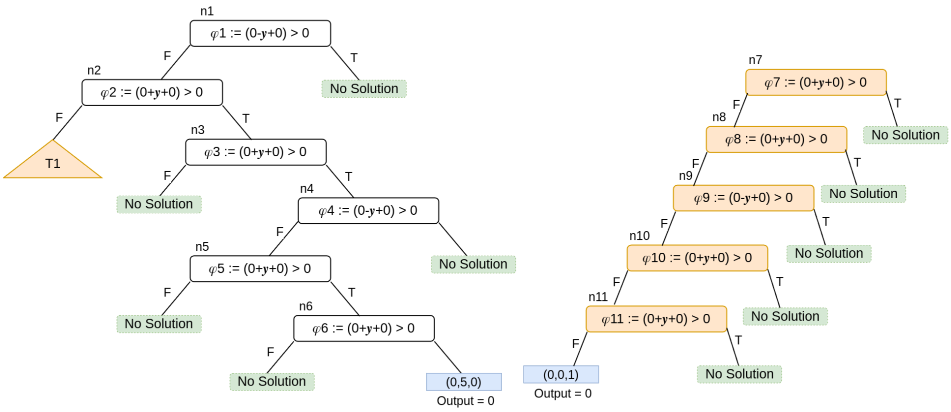

Fig. 5: Tree T (left) and Tree T' (right)

The 1^{st} *iteration:* We use $[0, 5, 0]$ as the initial input to explore the branch conditions. The T and Q collected by PyFair are displayed in Figure 5.

The 2^{nd} *iteration:* In this iteration, we proceed to examine the branches starting from the front of Q (Q7 in Figure 5). The SMT solver cannot find a solution, so we label the corresponding path as “no solution” (i.e., at the left branch of n6 in Tree T) in Figure 5.

The 3^{rd} *iteration:* In this step, we address the constraint $\neg\varphi_1 \wedge \neg\varphi_2$. The SMT solver generates a new test input $[0, 0, 1]$ and repeats the exploration process. The path condition explored by this third test input is depicted by Tree T’ in Figure 5. The prediction label for $[0, 0, 1]$ remains unchanged from the original. Consequently, we insert the values $x = 0, y = 0, x' = 1, Output = 0$ into the left branch of n11.

The 4^{rd} *iteration:* The process continues by dequeuing from Q. In this iteration, $\neg\varphi_1 \wedge \varphi_2 \wedge \neg\varphi_3$ has no solution, so we label “no solution” on the corresponding branch in the tree, similar to what was done in iteration 2.

The $5^{th} \sim 12^{th}$ *iterations:* Similarly, for the 5th to 12th iterations, PyFair continues by dequeuing Q to resolve constraints. Since the SMT solver returns UNSAT, indicating no solutions for each branch, we label “no solution” on the branches in Figure 5.

Once all path constraints are resolved (i.e., Q becomes empty), PyFair reports that no unfairness witness can be identified for the network. Since the network is expressible in SMT, this outcome certifies the network’s fairness with respect to the given PAs.

5. Evaluation

In our evaluation, we conduct discriminatory instance checking and model fairness verification for 25 models. We adopt a methodology similar to Fairify [18] and employ benchmark models from Fairify and previous studies [9]. These neural networks are fully connected, utilizing ReLU and Sigmoid as activation functions. Based on these models, we first evaluate the performance of discriminatory instance checking using PyCT for a single PA (RQ1) and PyFair for multiple PAs (RQ2). We also test models generated by existing bias mitigation techniques like ADF and EIDIG [8, 9], evaluating how effectively PyFair can identify discriminatory instances in well-trained models (RQ3). Finally, we investigate PyFair's efficacy in proving model fairness for perfectly fair models (RQ4).

We will use the following indicators in the tables: **UW** stands for whether unfairness is witnessed, with Y indicating a discriminatory instance is found, N indicating no evidence is found, and Unk meaning no conclusion within the time limit. **FQ** stands for the queue size after the initial sample's first forward pass in the model. The constraints within this queue are collected by modifying the states of activation functions. The FQ value therefore reflects the model's size. **#test** is the total number of samples tested within the time limit. **#sat** is the total number of constraints that the SMT solver determined to be SAT during the execution time. **#unsat** is the total number of constraints that the SMT solver determined to be UNSAT during the execution time. **Time** is the execution time. **Bias(%)** is employed to assess model fairness by determining the proportion of individual discriminatory instances within the dataset, and a lower percentage indicates greater fairness of the model.[2] **Fair** indicates the model fairness with Y for being fair, N for being unfair (i.e., a witness is found), and Unk for no conclusion within the time limit.

RQ1: What is the performance of PyCT in checking discriminatory instances on a single PA? In this experiment, we provide PyCT with 1500 random samples from the input domain. We evenly distribute these samples among 30 subprocesses for parallel processing. Each subprocess independently performs discriminatory instance checking on its assigned 50 samples. If any subprocess finds a discriminatory instance, the checking terminates and the time is recorded. We present the results below, where column "UW" indicates whether PyFair successfully identifies discrimination in the random samples.

			Fairify	PyCT					
PA	Model	Bias(%)	UW	UW	FQ	#test	#sat	#unsat	Time
Race	AC1	2.11	Y	Y	24	10	3	256	33.46
	AC2	0.8	Y	Y	100	2	1	117	33.55
	AC3	1.84	Y	Y	50	5	3	244	33.56
	AC4	1.06	N	Y	200	3	1	539	408.45
	AC5	2.48	Y	Y	128	1	1	79	106.82
	AC6	0.86	Y	Y	24	11	6	321	467.13
	AC7	0.78	N	Y	124	71	60	11082	2697.17
	AC8	1.89	Y	Y	10	95	25	950	175.38
	AC9	2.01	Y	Y	12	30	9	348	110.51
	AC10	1.54	Y	Y	20	11	1	213	33.61
	AC11	0.91	N	Y	40	7	1	250	1684.83
	AC12	1.54	N	Y	45	5	1	179	469.69

			Fairify	PyCT					
PA	Model	Bias(%)	UW	UW	FQ	#test	#sat	#unsat	Time
Age	BM1	0	Y	Y	80	138	5	11036	1053.78
	BM2	0	Y	Y	48	13	2	589	33.46
	BM3	0	Y	Y	100	58	1	5705	622.20
	BM4	0	Y	Y	300	590	20	147165	36028.09
	BM5	0.45	Y	Y	32	209	6	6512	388.63
	BM6	0.88	Y	Y	18	220	13	3822	332.68
	BM7	1.42	Y	Y	128	68	3	8366	1186.26
	BM8	0.52	N	Y	124	71	15	8976	2400.23
Sex	GC1	0.98	Y	Y	50	7	2	347	33.96
	GC2	2.07	Y	Y	100	2	1	145	68.03
	GC3	1.9	Y	Y	9	17	1	147	42.73
	GC4	0	Y	N	10	1500	0	15000	52217.74
	GC5	0	N	N	124	1443	6	169103	60809.37

We compare our tool with Fairify [18], a state-of-the-art constraint-based fairness verifier, over the same PAs on the same datasets. Fairify identifies discriminatory instances for 19 models with a timeout of 1800 seconds, as shown in column "Fairify UW". Detailed information from PyCT is also provided: "#test" indicating the total number of instances tested on the model (randomly selected from the dataset), and "#sat" and "#unsat" reflect the feasibility of path constraints, offering valuable insights into the overall viability of these constraints within the model. For instance, in the case of AC5, PyCT identifies its discriminatory instance right after the first test sample. It locates this instance after making 80 branch exploration attempts (the sum of #sat and #unsat). The SAT outcomes indicate the number of successful explorations, while the UNSAT ones indicate the visited hidden nodes that do not impact the output.

[2] More precisely, we repeatedly sampled 100 random inputs from the dataset and altered the PA value at random. If changing PA results in a classification shift, it is considered a discriminatory instance. We then compute the ratio of discriminatory inputs over 100 rounds. This calculation method is adapted from AEQUITAS [34].

We conclude that it is effective for PyCT to perform discriminatory instance checking. However, in some cases, it takes a long time to test a large number of samples, e.g., for GC4 and GC5, possibly due to insufficient sample diversity. In the subsequent experiment, we attempt to address this limitation using 2-DNN.

RQ2: What is the performance of PyFair in checking discriminatory instances on multiple PAs? In contrast to vanilla PyCT, PyFair is capable of checking discriminatory instances with multiple PAs thanks to the use of 2-DNN. For dual_ACs (i.e., the 2-DNN counterpart of the AC models), the runtime of checking multiple PAs is faster than checking a single PA in many tests. The "FQ" values of these tests indicate that the model complexity does not significantly increase with the number of PAs used. Moreover, the values of #sat and #unsat are generally lower, indicating that PyFair can more quickly identify discriminatory instances when multiple PAs are involved.

		PyFair				
PA	Model	UW	FQ	#sat	#unsat	Time
Race, Age, Sex	dual_AC1	Y	48	36	19	143.06
	dual_AC2	Y	200	1	0	47.8
	dual_AC3	Y	100	4	0	148.78
	dual_AC4	Y	400	20	1	333.41
	dual_AC5	Y	256	4	0	146.78
	dual_AC6	Y	48	27	25	1380.37
	dual_AC7	Y	248	13	3	3078.09
	dual_AC8	Y	20	18	1	16.36

		PyFair				
PA	Model	UW	FQ	#sat	#unsat	Time
Race, Age, Sex	dual_AC9	Y	24	9	6	11.94
	dual_AC10	Y	40	5	0	10.19
	dual_AC11	Y	80	27	4	1413.52
	dual_AC12	Y	90	6	1	1653.62
Age, Sex	dual_GC1	Y	100	166	134	41043.57
	dual_GC2	Y	200	6	2	171.08
	dual_GC3	Y	18	5	1	11.18
	dual_GC4	N	20	258	1833	1841.44
	dual_GC5	N	248	65	46	Timeout

Compared with vanilla PyCT, PyFair requires *only one* test sample to perform discriminatory instance checking, since PyFair can automatically generate all feasible test cases from a given sample. For example, PyFair exhaustively generates and tests 258 instance pairs for dual_GC4, while PyCT tests the GC4 model over 1500 random samples without identifying any new test case (see the table in RQ1). Although both approaches are inconclusive, dual_GC4 allows PyFair to explore all model branches within the time limit, offering higher coverage and confidence in fairness assessment. These results showcase PyFair's ability to handle multiple PAs simultaneously. On the other hand, PyFair still struggles with complex models like GC5, which indicates the challenges in thoroughly evaluating DNN fairness and the need of further optimization.

RQ3: How effective is PyFair in testing improved models by ADF and EIDIG? ADF and EIDIG [8, 9] exploit discriminatory instances to augment the data and retrain the model. Both retraining methods involve randomly sampling 5% of the generated discriminatory instances, relabeling them using majority voting [35], and incorporating these instances into the original training set before retraining the model on the augmented dataset. The effectiveness of these methods is shown in the left table of Table 2. As observed, models AC14, AC15, and AC16 exhibit lower Bias(%) compared to AC13. The difference between AC15 and AC16 lies in the frequency of recalculating gradients and attribute contributions. In AC15, these calculations are updated every five iterations (EIDIG-5), whereas in AC16 they are not updated (EIDIG-∞).

We test these four models using our fairness checking framework to evaluate how effectively PyFair identifies discriminatory instances in well-trained models. We observe that, even after applying ADF and EIDIG, the models are still vulnerable to discriminatory behavior. In Table 2, we present the results of using PyFair to analyze the original

Table 2: Checking Fairness of Biased Models (left) and Fair Models (right)

			PyFair				
PA	Model	Bias(%)	UW	FQ	#sat	#unsat	Time
Race, Age, Sex	AC13 (Original)	9.33	Y	180	20	6	30490.02
	AC14 (ADF)	2.61	Y	180	12	1	24475.46
	AC15 (EIDIG-5)	1.00	Y	180	8	1	15673.36
	AC16 (EIDIG-∞)	1.24	Y	180	12	1	22035.31

		PyFair				
PA	Model	Fair	FQ	#sat	#unsat	Time
Age	fair_dual_GC1	Unk	102	86	145	1800
	fair_dual_GC2	Unk	202	37	175	1800
	fair_dual_GC3	Y	20	1835	227	1798
	fair_dual_GC4	Y	22	382	28	40
	fair_dual_GC5	Unk	248	1	2	1800

model and the retrained models. Our tool identifies discriminatory instances for all four models. Interestingly, PyFair takes shorter time to identify discriminatory instances for the retrained models, which are statistically shown to be fairer than the original model. This result confirms the intuition that constraint-based methods are often more effective than sampling-based methods for detecting subtle biases.

RQ4: How effective is PyFair in proving model fairness? To answer this question, we evaluate PyFair's effectiveness on perfectly fair models. These models are derived from the benchmarks by setting the outgoing edge weights to zero for input nodes on PAs, ensuring that these attribute values do not influence the model outcome. Also, the Sigmoid function in the output layer is replaced with a direct comparison with the threshold, allowing PyFair to faithfully encode the network in linear real arithmetic.

The experimental results reveal both the potential and limitations of our tool in verifying model fairness, as only relatively small models can be fully verified within the 1800-second timeout. For example, in the right table in Table 2, the fair models verified by PyFair are relatively small (which can be seen by their small FQ values). For the GC5 model, PyFair only manages to explore one test case due to costly constraint solving. To conclude, even though PyFair can be employed to prove model fairness, the runtime can be considerable for complex models. This scalability issue underscores the need for more efficient techniques to handle the path constraints generated during fairness verification.

6. Conclusion

This work proposes a novel concolic testing framework for automatic fairness testing and verification. Our approach synthesizes sample inputs to explore different decision branches in model inference, achieving effective discriminatory instance verification and model fairness checking for pre-trained DNNs. The key strength of our framework lies in its dual network architecture and use of SMT solvers, which enable systematic exploration of fairness properties with formal guarantees for networks that can be encoded in SMT theories. However, just like other constraint-based testing approaches [15, 18], our evaluation reveals limitations around scalability, as the computational overhead becomes prohibitive for complex model architectures. Future work could enhance the framework through more efficient algorithms for handling larger architectures, extension to different network types like RNNs and Transformers, and deeper integration with fair model training techniques.

7. Acknowledgment

We thank the anonymous reviewers for their insightful comments. Chih-Duo Hong is supported by the NSTC, Taiwan, under grant number 112-2222-E004-001-MY3. Fang

Yu is supported by the NSTC under grant numbers 113-2221-E-004-010-MY2 and 113-2634-F-004-001-MBK.

REFERENCES

1. S. Biswas and H. Rajan, “Fair preprocessing: Towards understanding compositional fairness of data transformers,” in *Proc. 29th ACM ESEC/FSE*, 2021, pp. 981–993.
2. M. Hort, J. M. Zhang, F. Sarro, and M. Harman, “Fairea: Model behavior mutation for bias mitig.” in *Proc. 29th ACM ESEC/FSE*, 2021, pp. 994–1006.
3. J. M. Zhang and M. Harman, “Ignorance and prejudice in sw fairness,” in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1436–1447.
4. A. W. Flores, K. Bechtel, and C. T. Lowenkamp, “False positives, negatives, and analyses: A rejoinder to machine bias,” *Fed. Probation*, Vol. 80, 2016, p. 38.
5. J. Dastin, “Amazon AI recruiting tool bias against women,” *Ethics of Data and Analytics*. Auerbach, 2022, pp. 296–299.
6. S. Galhotra, Y. Brun, and A. Meliou, “Fairness testing: Testing software for discrimination,” in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 498–510.
7. P. Saleiro, B. Kuester, L. Hinkson, J. London, A. Stevens, A. Anisfeld, K. T. Rodolfa, and R. Ghani, “Aequitas: Bias and fairness audit toolkit,” *arXiv preprint arXiv:1811.05577*, 2018.
8. P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, “White-box fairness testing via adversarial sampling,” in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 949–960.
9. L. Zhang, Y. Zhang, and M. Zhang, “Efficient fairness testing via gradient search,” in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2021, pp. 103–114.
10. H. Zheng, Z. Chen, T. Du, X. Zhang, Y. Cheng, S. Ji, J. Wang, Y. Yu, and J. Chen, “Neuronfair: Interp. white-box fairness testing w/ biased neuron ident.” in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1–13.
11. T. Li, X. Xie, J. Wang, Q. Guo, A. Liu, L. Ma, and Y. Liu, “Faire: Repairing fairness via neuron synthesis,” *ACM Trans. Softw. Eng. Methodol.*, Vol. 33, 2023, pp. 1–24.
12. M. Feldman, S. A. Friedler, J. Moeller, C. Scheidegger, and S. Venkatasubramanian, “Certifying and removing disparate impact,” in *Proc. 21st ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2015, pp. 259–268.
13. C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, “Fairness through awareness,” in *Proc. 3rd Innov. Theor. Comput. Sci. Conf.*, 2012, pp. 214–226.
14. Y.-F. Chen, W.-L. Tsai, W.-C. Wu, D.-D. Yen, and F. Yu, “Pyct: A python concolic tester,” in *Proc. 19th Asian Symp. Prog. Lang. Syst.* Springer, 2021, pp. 38–46.
15. F. Yu, Y.-Y. Chi, and Y.-F. Chen, “Constraint-based adversarial example synthesis,” *arXiv preprint arXiv:2406.01219*, 2024.
16. K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated white-box testing of DNNs,” in *Proc. 26th Symp. Oper. Syst. Principles*, 2017, pp. 1–18.
17. H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, and A. Nötzli, “cvc5: A versatile and industrial-strength smt solver,” in *Int. Conf. Tools Algorithms Constr. Anal. Syst.* Springer, 2022, pp. 415–442.
18. S. Biswas and H. Rajan, “Fairify: Fairness verification of NNs,” in *2023 IEEE/ACM 45th Int. Conf. Softw. Eng.* IEEE, 2023, pp. 1546–1558.
19. A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, “Black-box fairness testing of ML models,” in *Proc. 27th ACM ESEC/FSE*, 2019, pp. 625–635.
20. T. Ige, W. Marfo, J. Tonkinson, S. Adewale, and B. H. Matti, “Adversarial sampling for fairness testing in DNNs,” *arXiv preprint arXiv:2303.02874*, 2023.

21. A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: Probabilistic verification of program fairness," *Proc. ACM Prog. Lang.*, Vol. 1, 2017, pp. 1–30.
22. O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness via concentration," *Proc. ACM Prog. Lang.*, Vol. 3, 2019, pp. 1–27.
23. P. G. John, D. Vijaykeerthy, and D. Saha, "Verifying individual fairness in ML models," in *Conf. Uncertainty in AI*, 2020, pp. 749–758.
24. A. Ruoss, M. Balunovic, M. Fischer, and M. Vechev, "Learning individually fair reprs." *Adv. Neural Inf. Process. Syst.*, Vol. 33, 2020, pp. 7584–7596.
25. C. Urban, M. Christakis, V. Wüstholz, and F. Zhang, "Perfectly parallel fairness cert. of neural nets," *Proc. ACM Program. Lang.*, Vol. 4, no. OOPSLA, 2020, pp. 1–30.
26. M. Yurochkin, A. Bower, and Y. Sun, "Training individually fair ML w/ sensitive subspace robustness," *arXiv preprint arXiv:1907.00020*, 2019.
27. H. Khedr and Y. Shoukry, "Certifair: Certified global fairness of NNs," in *Proc. AAAI Conf. Artif. Intell.*, Vol. 37, 2023, pp. 8237–8245.
28. K. Mohammadi, A. Sivaraman, and G. Farnadi, "Feta: Fairness enforced NN algorithms," in *Proc. 3rd ACM Conf. Equity Access Algorithms Mech. Optim.*, 2023, pp. 1–11.
29. Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for DNNs," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 109–119.
30. L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, and Y. Liu, "Deepgauge: Multi-granularity testing for DNN systems," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 120–131.
31. Z. Zhou, W. Dou, J. Liu, C. Zhang, J. Wei, and D. Ye, "Deepcon: Contribution coverage testing for DNN systems," in *IEEE Int. Conf. Softw. Anal., Evol. and Reeng.* IEEE, 2021, pp. 189–200.
32. Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: Testing and debugging DNNs," in *IEEE/ACM Int. Conf. Softw. Eng. Companion.* IEEE, 2019, pp. 111–114.
33. L. De Moura and N. Bjørner, "Z3: Efficient smt solver," in *Int. Conf. Tools and Algorithms for Constr. and Anal. of Syst.* Springer, 2008, pp. 337–340.
34. S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *Proc. 33rd ACM/IEEE ASE*, 2018, pp. 98–108.
35. L. Lam and S. Y. Suen, "Majority voting in pattern recognition," *IEEE Trans. Syst. Man Cybern. A: Syst. Hum.*, Vol. 27, no. 5, 1997, pp. 553–568.