



Applying Artificial Neural Networks to Logic Built-in Self-test: Improving Test Point Insertion

Yang Sun^{1,2} · Spencer K. Millican¹

Received: 16 November 2021 / Accepted: 15 July 2022 / Published online: 4 August 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

This study applies artificial neural networks (ANNs) to increase stuck-at and delay fault coverage of logic built-in self-test (LBIST) through test point insertion (TPI). Increasing TPI quality is essential for modern logic circuits, but the computational requirements of current TPI heuristics scale unfavorably against increasing circuit complexity, and heuristics that evaluate a TPs quality can mask the effects of delay-causing defects that are common in modern technologies. Previous studies show ANNs giving substantial benefits to a wide array of electronic design automation (EDA) problems, including design-for-test (DFT), but their application to various DFT problems is in its infancy. This study demonstrates how to train an ANN to evaluate test points (TPs) and demonstrates a substantial decrease in TPI computation time compared to existing heuristics while delivering comparable stuck-at and delay fault coverage.

Keywords Artificial neural networks · Built-in self-test · Stuck-at faults · Delay faults · Test point insertion

1 Introduction

For modern technologies, creating high-quality digital tests is a challenge that, if fulfilled, provides several advantages to circuit developers and manufacturers. Semiconductor manufacturing is an imperfect process despite decades of semiconductor research [15], and the release of defective devices has both economic (e.g., loss of reputation) and catastrophic (e.g., loss of life) consequences. This latter consequence is of growing concern for modern technologies, since devices are more frequently deployed in life-critical applications (e.g., self-driving cars and implanted medical devices). For these reasons, creating high-quality manufacturing tests is necessary to detect and to prevent the release of defective circuits or to fuse-off defective portions and release partially-good circuits. Additionally, high-quality tests may need to be applied in-the-field to ensure continuing circuit

reliability: aggressive technology scaling accelerates post-manufacturing degradation and the occurrence of intermittent soft errors [21], and like pre-delivery tests, tests must detect (and possibly remedy) these in-field defects.

Logic built-in self-test (LBIST) is an ideal test mechanism for delivering high-quality manufacturing and post-manufacturing tests. LBIST uses on-chip stimulus generators, typically in the form of *pseudo-random pattern generators* (PRPGs), to stimulate and set the state of logic circuits while observing circuit responses. The logic circuit industry uses LBIST to apply manufacturing tests since LBIST can obtain significant fault coverage while decreasing test application time, which in turn reduces manufacturing test costs. Additionally, in-field tests can use LBIST since embedded LBIST circuits can test circuits with minimal functional interruptions by saving the circuit state, applying test-enabling signals, and then reloading the circuit state to resume normal circuit operation.

However, keeping LBIST quality high is challenging under aggressive technology scaling since logic circuit complexity is growing faster than *design-for test* (DFT) techniques can remedy. LBIST tests have difficulty detecting *random pattern resistant* (RPR) faults (see Sect. 2.1), which is unfortunate since these faults become more common as logic circuitry becomes more complex. Many DFT techniques attempt to detect RPR faults during LBIST, such as weighted random pattern testing [27] and

Responsible Editor: E. Amyeen

✉ Spencer K. Millican
millican@auburn.edu

Yang Sun
yzs0057@auburn.edu

¹ Department of Electrical and Computer Engineering, Auburn University, Auburn, AL 36849, USA

² Marvell Technology, Inc., Santa Clara, CA 95054, USA

deterministic pattern seeding [17]. One particular technique, *test point (TP) insertion (TPI)* [1, 11, 19, 22, 26], is the focus of this article, since TPI can be used in conjunction with other techniques to further increase LBIST quality. Unfortunately, the computational complexity of the algorithms that implement TPI increase faster than the size of logic circuits [24]. Since computational resources are in high demand by several *electronic design automation (EDA)* tool users during circuit development, designers must sacrifice testability or other circuit qualities if EDA tool designers do not increase algorithm efficiency.

New computing methods, such as *artificial neural networks (ANNs)*, can increase algorithm efficiency and keep LBIST quality high. ANNs can solve complex problems, such as image and speech recognition, and significantly increase the quality of existing algorithms while simultaneously decreasing computation time. Recently, studies applied ANNs to several EDA problems with noteworthy success [11], but applying ANNs to DFT problems is in its infancy.

This article demonstrates how ANNs can increase TP quality when applied to TPI while drastically decreasing algorithm runtime. This article is the accumulation of several years of work [13, 22] which made the following contributions:

1. The creation, training, and use of an ANN that evaluates the stuck-at fault detection effectiveness of TPs (which this article further improves), and a comparison of TPI stuck-at fault coverage against heuristic-based TP evaluation.
2. The creation, training, and use of an ANN that evaluates the delay fault coverage impact of a TP, and a comparison of delay fault coverage when used in TPI against heuristic-based TP evaluation.
3. A thorough run-time analysis of TPI and a conclusive demonstration of an ANN's ability to reduce TPI runtime by orders of magnitude.

Beyond the accumulation of previous studies, this article makes the following contributions not seen in previous literature:

1. The ability to train a TP-evaluating ANN to consider the number of LBIST vectors, which makes the method more applicable to typical LBIST uses.
2. An improvement to the TP-evaluating ANN's output label based on its meaning and relevance during an LBIST test.
3. A detailed exploration of the impact of ANN complexity; experiments compare three ANNs with different feature sizes and observe the impact of increased accuracy in lieu of possible longer computation time.

The remainder of this article is organized as follows. Section 2 provides background and motivation for this article's contributions to TPI. Section 3 presents extensions to previously published work [22], which is an ANN used for stuck-at fault detection, and Sect. 4 shows how a similar ANN can be used for delay fault detection. Section 5 shows how ANN training and hyperparameters – i.e., the structure and complexity of the ANN – can significantly affect an ANN's quality. Section 6 evaluates the ANN-based TPI method compared to equivalent heuristics, and Sect. 7 concludes this article and proposes future research directions.

2 Background & Motivation

2.1 Test Points

In the context of LBIST, TPs aim to make detecting RPR faults easier for circuits under pseudo-random stimuli. A typical example of an RPR fault is the output of a 64-bit wide OR gate (e.g., a global error-detecting circuit which polls 64 individual error lines) stuck-at logic-1. Exciting this fault requires driving the faulty line to logic-0, which requires every input to the OR gate to be logic-0. Under truly random stimulus (i.e., every input to the OR gate has a 50% chance of being logic-0/1), the probability of this input vector occurring is $2^{-64} \approx 0$.

To make RPR faults easier to detect, TPs make circuit lines either (1) easier to control or (2) easier to observe. In the above example, when forcing a subset of input pins to logic-0 during test, the probability of exciting and detecting the fault doubles for every pin forced.

Several implementations of LBIST TPs exist in literature, but the typical implementation consists of control points and observe points. Intuitively, a multiplexer can force circuit lines during test: a *test enable (TE)* signal controls the multiplexer, which leaves the circuit function intact when inactive but forces a desired value when active. This implementation is wasteful: circuit lines requiring direct control typically need only a single value forced since the other value is already easy to obtain. Alternatively, TE can force the desired value with a single gate, as illustrated in Fig. 1, thus making a control point. Likewise, a new circuit output can be added to hard-to-observe locations, thus making an observe point. Since a line may be simultaneously hard-to-control and hard-to-observe, such a line may need both a control and observe point.

The additional signals for control/observe points can either be pins or scannable latches, and tests do not need to activate all control points simultaneously. Although algorithms typically model TE and observe points as pins for simplicity (similar to how scannable latches are typically

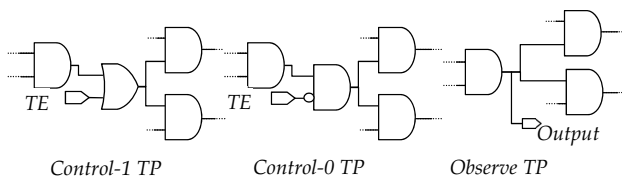


Fig. 1 The typical implementation of LBIST TPs. Control TPs require an extra test enable (TE) signal and observe points require an extra output pin or scannable latch

modeled as circuit inputs and outputs by test generators), implementing them as circuit pins is impractical given their high cost. Instead, scannable latches implement these “pins”: observe points feed the input to latches and TE is the output of a latch, and tests exclusively use these scannable latches. When using such latches, it is possible to have multiple TE signals for different sets of control points, and activating sub-sets of TE signals can increase TP effectiveness [18, 25]. The effectiveness of partial control point activation is not in the scope of this article; this study presumes that a TP method that is effective for universal control point activation will likewise be effective for partial control point activation.

2.1.1 Problem – Computational Difficulties

The challenge of TPI is to place the fewest number of TPs while maximizing fault coverage. Each TP requires logic circuitry, which in turn creates the undesirable overheads of static and dynamic power, delay, and non-functional circuit area. Designers typically allocate budgets to TPs (and other DFT hardware), thus TPs must increase LBIST fault coverage to acceptable levels while enforcing a TP hardware budget.

The problem of selecting optimal TP locations (and many other DFT problems) is a known NP-hard problem [24], thus existing TPI methods rely on heuristic approaches to select TP locations. Inserting T TPs into a circuit among T' candidate TPs creates $C(T', T)$ possible TP choices, and finding the fault coverage impact of a choice requires computationally-intensive fault simulation. TPI heuristics address this by replacing fault simulation with less accurate fault coverage estimations [6, 26] while simultaneously using greedy-algorithm approaches. The most common approach is iterative [11, 19, 26]: heuristically evaluate candidate TPs to find the one which increases fault coverage the most, insert it, and repeat this process until no more TPs are desired or needed.

Unfortunately, the computational complexity of iterative TPI grows faster than available computing resources. Let G be the growth in a size of a circuit, i.e., the number of gates in a circuit that grows with every technology generation. To insert a single TP, every candidate TP’s impact on fault coverage must be evaluated to find the best one, and the number

of candidate TPs in a circuit is proportional to the size of the circuit, i.e., one or more candidate TPs can exist on every gate input/output (or in equation form, $T' \sim G$): this creates a CPU run-time complexity of $O(G)$ to select a single TP if the time to calculate a TP’s quality is constant. However, the complexity of TP-evaluating heuristics is not constant: heuristics must evaluate the TP’s effect on fault coverage on every gate in the circuit, which itself is an $O(G)$ time calculation. Although TPI methods attempt to minimize this time by only evaluating areas of the circuit effected by the TP (e.g., up/down-stream of the TP [26]), such areas can still encompass the majority of the circuit. Likewise, simulation-based TPI methods [23] still require some degree of re-simulation with the TP in-place to find the TP’s true impact on fault coverage. Besides increasing the time to evaluate a TP’s impact on fault coverage, increasing circuit complexity also introduces more circuit faults (denoted as $F \sim G$), and thus requires more TPs to obtain reasonable fault coverage. All these factors put together implies TPI CPU time grows with respect to $O(T \cdot G \cdot F)$, and since all of these variables are related to circuit complexity, this can be simplified as $O(G^3)$. Even if computing resources grows with respect to circuit complexity (i.e., more complex circuits give faster computers) – thus dividing this growth by G – the CPU time still grows at a quadratic rate of $O(G^3/G) = O(G^2)$.

2.1.2 Problem – Delay Defect Masking

As newer technologies implement logic circuits, tests must model and detect nuanced defects, most notably delay defects. Tests must still detect stuck-at faults, but technology scaling makes delay-causing defects more common [16]. To detect these defects, LBIST (and other tests) must perform at the circuit’s designed clock speed and create the transitions necessary to excite and propagate failing transitions to observable locations. For these reasons, LBIST is an ideal technique for delay fault testing, as applying tests with chip-internal hardware eliminates the need for slow external pattern generators.

Many studies demonstrated TPs effectively testing stuck-at faults, but control points can lower delay fault coverage when TPI targets stuck-at faults. Delay faults require transitions on faulty lines to both excite and propagate a fault’s effect to observable locations, but this is impossible when control points are active: control points force constant values, and therefore transitions can never occur on a controlled line, as illustrated in Fig. 2.

Ad hoc remedies can improve delay fault coverage when using TPs, but such approaches create undesirable complications. Existing TPI methods may forbid control points (i.e., control points will not be in the candidate TP list), but this is impractical. First, control points require less area and power overhead compared to observe points (one gate and a shared

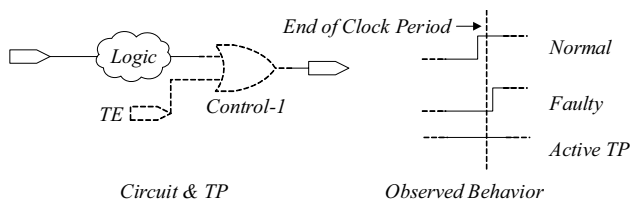


Fig. 2 Active control points force values and block transitions: this prevents delay-causing defects from exciting and propagating to observable locations

TE latch vs. one latch per observe point). Second, control points may be the best TP choice for some circuits, since some faults (stuck-at or delay) may require forced circuit lines, which observe points cannot create.

2.2 Artificial Neural Networks

An ANN is a model of a biological neural network (i.e., a brain) in software or hardware. ANNs excel at solving problems when manually-developed heuristic algorithms falter. ANNs “learn” by providing problems with known solutions (i.e., training data) and “train” themselves until correct/accurate results are achieved for each training problem. In the past, ANNs were infeasible to implement due to a lack of training data [8], but many industries today use ANNs, with handwriting and speech recognition being notable examples.

Creating and training ANNs is described in other sources [8], but below is a brief summary. For readers with an EDA/DFT background, some ANN terms can cause confusion since they are used in common in English but have a particular meaning in ANN research; for clarity, such terms are highlighted in *italics* below.

Creation (ANN Structure Selection) Many ANN structures exist in literature, but Fig. 3 illustrates the prototypical ANN structure. ANNs contain *neurons*, and some neurons are input *feature* and output *label* neurons. *Dendrites* multiply (*weigh*) and add/subtract (*bias*) neuron outputs as well as connect the neurons. Neurons have *activation functions* that calculate the neuron output based on neuron inputs.

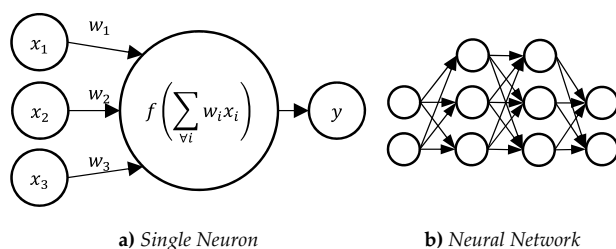


Fig. 3 An example of **a** a single neuron with input signals, input weights, and an activation function, and **b** a neural network composed of multiple layers

Many choices exist for neuron activation functions and neuron arrangements (e.g., the number of levels and neurons per level), and these *hyperparameters* are best optimized through trial-and-error.

Training Training finds weights and biases such that a set of example inputs will provide desired outputs. Depending on the ANN configuration, the number of features and labels, and the amount of training data, this can be computationally-intensive and is not an optimal process. Time permitting, repeated training using various hyperparameters will optimize weights, biases, and the ANN topology.

Recently, studies applied ANNs of various forms to DFT testing problems, including scan-chain diagnosis [4] and fault diagnosis [7], but few studies applied ANNs to TPI. Ma et al. [11] applied ANNs to TPI, but this study limited its scope and application. The study used observe points exclusively, analyzed only four benchmarks, did not analyze CPU time, and modeled only stuck-at faults.

3 ANN for Stuck-at Fault TPI

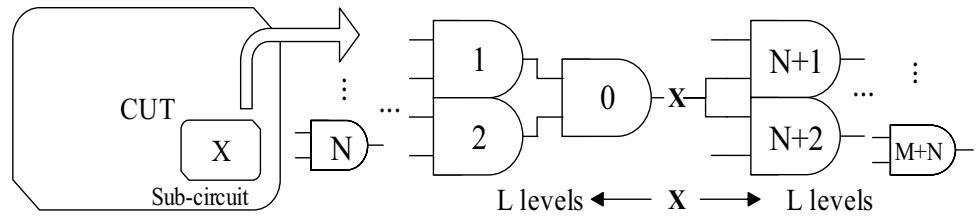
This section describes the function, training, and use of an ANN for increasing stuck-at fault coverage through TPI while also minimizing TPI CPU time. This ANN is an enhanced version of the ANN described in [22]: modifications to the ANN allow (1) more accurate predictions of TP quality under varying amounts of random stimulus and (2) more relevant fault coverage impact metrics to further enhance TP quality.

The final result of training will be several different ANNs. Each ANN will predict a single TP type’s (i.e., control-0, control-1, or observe) impact on a type of fault coverage (i.e., stuck-at fault coverage, with the next section’s ANNs predicting the impact on transition delay fault coverage). For example, one ANN will judge the impact a control-1 TP has on stuck-at fault coverage, and a separate ANN will judge the impact an observe TP has on delay fault coverage. Although it is possible to train a single ANN with multiple outputs, this would increase training time and complexity. Once the ANNs are trained, it can be re-used for any circuit without re-training. The final configuration of ANNs is described in Sect. 6.1.

3.1 Function, Use, and Input Features

The proposed ANNs evaluate TPs and can be used in any iterative TPI algorithm [11, 19, 26]. In each iteration, the algorithm evaluates every individual TP to find the “best” TP (i.e., the TP which increases fault coverage the most), and the algorithm inserts this TP into the circuit. This

Fig. 4 Since the input sizes of the TP-evaluating ANNs are constant, training and ANN evaluation extracts subcircuits with a two-input, two-fanout structure. Here, “CUT” represents circuit under test, “X” marks the TP location



iterative selection continues until (1) the number of TPs inserted reaches a pre-designated limit (representing hardware overhead), (2) the predicted fault coverage reaches a pre-designated limit (i.e., no more TPs are necessary), (3) no TPs are predicted to increase fault coverage, or (4) a CPU time limit is reached.

The ANNs evaluate TPs using circuit probability information, i.e., the same *controllability-observability program* (COP) values [2] that many other TPI methods use [11, 19, 26], but unlike other TPI methods, the ANNs do not require recalculating this information when evaluating a TP. Every TP selection requires recalculating these values only once, but evaluating a single TP does **not** require recalculating COP values. This is a noteworthy potential advantage of the ANNs over TP-evaluating heuristics that require recalculating values when evaluating every TP (see Sect. 2.1.1), since calculating COP values for a circuit with G gates requires $O(G)$ time (although some algorithms attempt to minimize unnecessary recalculations [26]). Performing COP once per TP selection as opposed to once per TP evaluation reduces the time to evaluate a single TP from $O(G)$ to $O(1)$, which reduces the time to select the best TP amongst T TPs from $O(T \cdot G)$ to $O(T)$. However, for ANN-based TPI, long training times might negate this benefit. The overall effect on TPI time (with and without training) will be explored in Sect. 6.3.

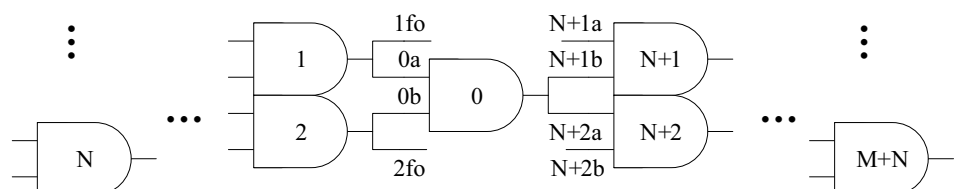
In contrast to conventional TP-evaluating heuristics, the ANNs perform their evaluation on a transformed sub-circuit centered around a candidate TP location as opposed to estimating a TP’s impact on the entire circuit. This is a consequence of ANNs’ rigid structure: the input size of an ANN must be of a set size, whereas logic circuits can be of

any size and topology. In theory, the ANNs can analyze an entire circuit, but this requires infeasibly large ANNs that are impossible to train. Instead, the ANN analyzes features around a TP’s location by analyzing a subcircuit L levels forwards and backwards of the indicated location, as Fig. 4 illustrates.

Using a subcircuit to analyze a TP’s impact on an entire circuit presents a potential detriment. The ANNs use less information for their TP-evaluating calculations and may return less accurate qualifications. The ANNs will be as accurate (or more) as a heuristic using identical input features only if L is large enough to capture the entire circuit, but this creates infeasibly large ANNs that are impossible to train. Section 6.2 explores whether analyzing subcircuits decreases the ANNs’ performance compared to heuristics.

A nuance of the ANN subcircuits is they must have a particular circuit configuration (hence “transformed” in transformed subcircuits): every gate must have at most two inputs and two fan-outs. The ANN input features represent values at $N + M$ locations going backwards and forwards, respectively, through L circuit levels, with each location in the array representing the features at a gate. Figure 5 illustrates this feature array and the locations corresponding to each array location. The training algorithm and TP-evaluating subroutine transforms circuits that are not in this two-input, two-fanout form. The transformation replaces gates with more than two inputs with a tree of gates that implement the same function, and it replaces nets with more than 2 fan-outs with trees of buffers. For gates with one input or one fan-out, “default values” replace missing values in the feature string: non-existent lines have 50% controllability and 0% observability, and non-existent gates are replaced

Fig. 5 The ANN input features are the COP controllabilities (CC), COP observabilities (CO), one-hot encodings of gate types (*Gate*) before and after the TP location, and the number of LBIST test vectors (V)



NN Input Vector

$CC_0, CC_{1fo}, CC_{0a}, CC_{0b}, CC_{2fo}, CC_1, CC_2, \dots, CC_{Nb}, CC_{N+1a}, CC_{N+1b}, \dots, CC_{N+M-1}, CC_{N+M},$
 $CO_0, CO_{1fo}, CO_{0a}, CO_{0b}, CO_{2fo}, CO_1, CO_2, \dots, CO_{Nb}, CO_{N+1a}, CO_{N+1b}, \dots, CO_{N+M-1}, CO_{N+M},$
 $Gate_0, Gate_1, Gate_2, \dots, Gate_N, Gate_N+1, Gate_N+2, \dots, Gate_N+M, V$

with a none-hot encoding as opposed to a one-hot encoding (e.g., 0001 = AND, 0010 = OR, ..., 0000 = no gate). During TPI, only TPs on lines present in the original circuit will be evaluated and inserted.

The final ANN input is the number of pseudo-random vectors applied during LBIST, which is an addition to previous studies [22], which presumed every test applied ten-thousand vectors. Depending on the number of LBIST vectors to apply, the optimal location of TPs can change. To account for this, this study appends the number of vectors to apply, V , to the ANN feature string and expands the training data generation process (see the next section) to train the ANN with this additional feature. Industrial users can benefit by training their ANN with a wide range of V values representative of typical industrial practices – e.g., 1 K, 256 K, and 1 M vectors – and should gather many such training vector ranges as part of their large training databases. This academic study constrains itself to fewer than ten-thousand vectors.

3.2 Training and Output Labels

Training the ANN is a two-step process: first, fault simulation with TPs obtains training data; second, a training algorithm uses this data to train the ANN while exploring hyperparameters to maximize accuracy. This section describes both processes.

The ANN's output label is the number of additional (or possibly fewer) faults detected in the subcircuit when inserting a TP into a subcircuit's center "X", as shown in Fig. 4. This label contrasts with the label from previous studies [22], which was the relative change in fault coverage in the subcircuit. This new label more accurately models the impact of partial subcircuits on the entire circuit: TPs on separate subcircuits can have the same impact on subcircuit fault coverage (e.g., +10%) while detecting a different number of faults because one subcircuit is filled with more default values that do not contain faults to detect, which makes the TP that detects more faults the better choice. If a TP will decrease the number of faults detected (i.e., by masking faults), the label is a negative number.

Using the change in the number of faults detected in a subcircuit, as opposed to the number of faults detected in the entire circuit, poses a challenge: the quality measure may not adequately represent the TP's impact on the entire circuit. It is possible that many faults will be detected in the subcircuit but fewer faults will be detected outside the sub-circuit and vice versa. Section 6.2 explores if this potential detriment has a negative impact on fault coverage after TPI.

Fault simulation collects training data for the ANNs, but additional techniques must reduce training data collection time. Conceptually, simulating V vectors, inserting a TP

(randomly or deterministically), and repeating fault simulation to find the change in the number of faults detected gives a TP's true impact on fault coverage, and this result is more accurate than any heuristic-based methods that use less accurate fault coverage calculations [2]. However, fault simulation is computationally demanding: fault simulating V vectors in a circuit with G gates and F faults requires $O(V \cdot G \cdot F)$ time, therefore collecting S training samples requires $O(S \cdot V \cdot G \cdot F)$ time. To collect this training data, reducing V is not an option, since LBIST typically applies many vectors. This study attempted to collect training data through the direct application of vectors to circuits, but in reasonable overnight runs, only hundreds of training samples were collected, and the resulting ANNs selected TPs that consistently failed to increase fault coverage regardless of training effort.

The first training speedup technique is to apply fault simulation to subcircuits with assistance from circuit probability calculations (i.e., COP values). Compared to performing fault simulation on an entire circuit, performing fault simulation on subcircuits significantly reduces both G and F since the number of faults in a subcircuit is proportional to the number of gates, which in turn reduces fault simulation time. However, applying random subcircuit inputs and directly observing subcircuit outputs is not realistic: under (pseudo-) random circuit inputs, subcircuit inputs are not truly random, nor are subcircuit outputs always observed. To account for this, the training data generation program first calculates COP controllability and observability values [2] once per training circuit. This additional one-time $O(G)$ calculation time is negligible when taking a significant number of subcircuit samples from a circuit. Then, fault simulation weighs each subcircuit input vector using these COP controllability values. Additionally, if a fault's effect reaches a subcircuit output, fault simulation probabilistically detects it using the COP observability values of subcircuit outputs. Although this technique significantly decreases training data generation time, the technique may hinder the ANNs' ability to select high-quality TPs since subcircuit controllability and observability values are not one hundred percent accurate [2]. Section 6.2 explores this potential detriment.

A second technique reduces training data generation time by eliminating redundant vectors. In this study, subcircuit sizes are small enough that applying V vectors will guarantee redundant vectors: the number of possible vectors to apply to a circuit with I' inputs is $2^{I'}$, and $V \gg 2^{I'}$. Subcircuit input probabilities exacerbate this, since they make some vectors more likely to occur than others. To remedy this, the training data generation program applies each vector v' among the $2^{I'}$ possible subcircuit input vectors at most once. This is done by calculating the probability a vector will be applied to a sub-circuit if V vectors are applied to the entire circuit, denoted as $p_V(v')$. COP

values can calculate the probability a sub-circuit vector is applied with a single random circuit vector, denoted as $p_1(v')$, which in turn can find the probability of not applying the vector among V vectors, denoted as $(p_V(v'))$. This is calculated using the following equation: v'_i are the binary values for each subcircuit input in v' , and $CC_i(v'_i)$ is the probability this value will occur, which is calculated using circuit-wide COP values using the following equation.

$$p_V(v') = 1 - \overline{p_V(v')} = 1 - (1 - p_1(v'))^V$$

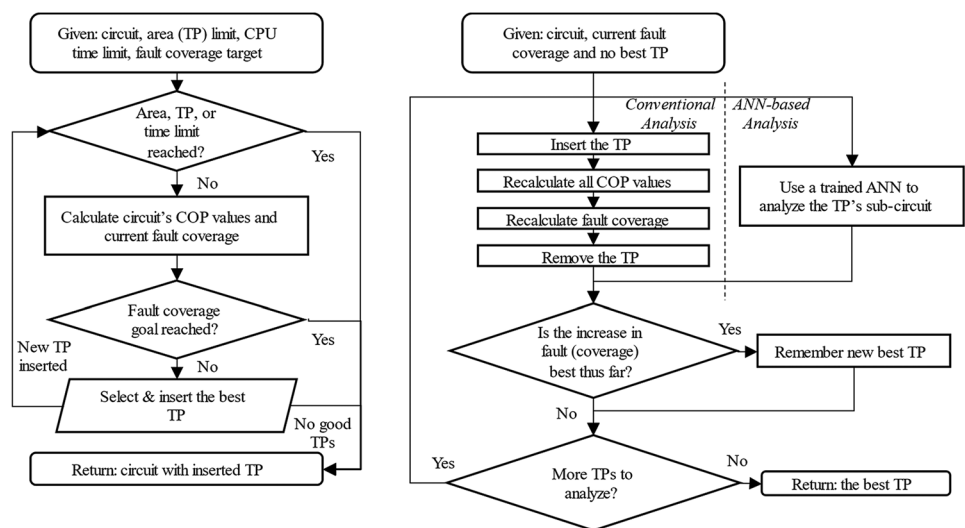
$$= 1 - \left(1 - \prod_{v_i \in I'} CC_i(v'_i)\right)^V$$

After creating the training data, the collected data trains an ANN under various ANN hyperparameters to minimize ANN error (in this study, *mean squared error*, or MSE). Finding a truly optimal ANN for a given training data set is an NP-hard problem [8], and small changes in initial training conditions and hyperparameters can significantly impact the resulting ANN's quality, thus a trial-and-error process minimizes this error. Section 5 explores the effect of different ANN hyperparameters on the ANN error.

3.3 ANN Use in TPI Flows

The use of a trained ANN in a TPI algorithm is analogous to the use of a TP-evaluating heuristic in an iterative TPI algorithm [26]. A trained ANN calculates a candidate TP's quality, and after doing so for every candidate TP, the highest quality TP is inserted. A comparison of TPI flows is shown in Fig. 6.

Fig. 6 The flow of ANN-based TPI is much like other LBIST TPI algorithms, except the subroutine which evaluates a TP's impact on pseudo-random fault coverage is replaced with a trained artificial neural network. Note that once the network is trained, it can be re-used for any number of circuits



(a) Iterative TPI

(b) TP evaluation (conventional vs ANN-based)

4 ANN for Delay Fault TPI

As Sect. 2.1.2 discussed, TPs can block delay faults from propagating through a circuit or from being excited, and TPI should try to prevent this. This section's TP-evaluating ANNs presume using observe points exclusively (as discussed in Sect. 2.1.2) is unacceptable to designers. Therefore, these ANNs target TP locations that increase delay fault coverage as much as possible when using conventional TP architectures (i.e., both control and observe points).

To apply the techniques from Sect. 3 to delay fault coverage, the ANNs' output labels must change from the number of stuck-at faults detected to the number of delay faults detected. This change is a two-step process. First, the training data collection program uses the *transition delay fault* (TDF) model in lieu of the stuck-at fault model. Alternative delay fault models exist, but studies show the TDF model represents industrial faults, and implementing the TDF model requires minimal changes to stuck-at fault simulators [23]. Second, instead of probabilistically simulating $2^{I'}$ vectors for each subcircuit, the training data generation program probabilistically simulates $2^{2 \cdot I'}$ vector pairs because delay faults require two vectors to detect: one to set the initial state and another to launch the circuit transition. Like the probability of applying a single vector, the probability of applying a vector pair $\{V_1, V_2\}$, can be calculated: $p_v(v_1) \cdot p_v(v_2)$. Also, since $V \ll 2^{2 \cdot I'}$, all vector pairs are still simulated (and faults observed) at most once.

Like the stuck-at fault-qualifying ANNs, the delay fault-qualifying ANNs train under various ANN hyperparameters to minimize error. The hyperparameters of these ANNs may not be the same as the stuck-at fault-qualifying ANNs, but to allow for a fair comparison of the two, the following

sections' evaluations use the same amount of training time for both ANNs. This presents a potential detriment as generating a training data sample takes more time for the delay fault-targeting ANN, as fault simulation applies $2^{2'}$ vector pairs per sample instead of $2'$. Section 6.2 explores the final effect of this potential detriment.

5 ANN Training Experiments

This section explores the impact hyperparameters have on ANN quality, while the next section determines the ANNs' final ability to evaluate a TP. As previous sections noted, hyperparameters greatly affect the quality of an ANN, which includes the ANN topology (convolutional, deep neural networks, fully/sparsely connected, etc.), the number of neurons, the number of hidden neuron layers, and activation functions. Additionally, training parameters and effort significantly influence ANN quality, e.g., step size (how much to change dendrite weight and bias magnitudes each iteration) and initial conditions.

Given the vast parameter exploration space, this article limits itself to showing the exploration of two hyperparameters: training data set size and the number of ANN neurons. In these explorations, all other hyperparameters are constant. These constant hyperparameters are the neuron arrangements (two hidden layers, with variable amounts of neurons in the first hidden and a single neuron in the second hidden layer, and neurons using sigmoid activation functions [12]) and training parameters (using the Adam optimization algorithm [10] with a 0.01 training step size). Training performs 55,000 iterations for each ANN. For every ANN in this study, ANN error stops decreasing before these iterations are complete.

5.1 Training Data Set Size

A known roadblock to previous ANN implementations is the lack of available training data [8]: if there's not enough training data to find a correlation between input features and desired output labels, then creating a useful ANN is impossible. However, providing too much training data increases training time and can degrade ANN quality through "overfitting" [9]. For these reasons, this article explores how much data is required to minimize ANN error. Whether this ANN is useful at evaluating a TP is best explored after training (see Sect. 6).

A random selection of ISCAS'85 [3] and ITC'99 [5] benchmarks serve as training circuits, and ANN training experiments are based on these benchmarks. ISCAS'89 benchmarks were not used since the combination of the ISCAS'85 and ITC'99 benchmarks provide sufficiently diverse sizes of circuits. Table 1 lists these benchmarks,

Table 1 Training Benchmark Circuits

<i>Benchmark Circuit</i>	Inputs	Outputs	Gates	Training Samples
c17	5	2	13	2
b02	5	5	32	2
b06	11	15	65	2
b08	30	25	204	4
b10	28	23	223	5
c499	41	32	275	6
c1355	41	32	619	12
b04	77	74	803	16
b12	126	127	1,197	26
c2670	233	140	1,566	25
c6288	32	32	2,480	50
c7552	207	108	3,827	72
b14_1	277	299	7,145	165
b15_1	485	519	13,547	311
b21_1	522	512	14,932	345
b20_1	522	512	14,933	342
b22_1	767	757	22,507	516
b17_1	1,452	1,512	41,080	949
b18_1	3,357	3,343	111,802	2,520
b19_1	6,666	6,672	226,066	5,087

their physical qualities (the number of gates, inputs, and outputs), and the number of randomly-extracted training samples per circuit. The number of samples extracted per circuit is proportional to the size of each circuit, i.e., larger circuits have more samples extracted from them. In the table, circuit *Inputs* and *Outputs* include the outputs and inputs of latches (respectively); this study presumes circuits are tested in a full-scan environment and thus latches are fully observable and controllable during test. Note that some circuits are small and easy to test with pseudo-random stimuli; this is desirable for generating training data, since bad/useless TPs must be accurately evaluated as much as good TPs in order to avoid choosing them during TPI.

Figure 7 shows the impact different training data sizes have on a stuck-at fault-targeting, control-1 TP-evaluating ANN's error. Each horizontal point corresponds to selecting a different number of randomly-selected training samples: 2,097 samples, 10,457 samples (used for training the ANNs in Sect. 6, and samples from individual circuits are given in Table 1), 20,910 samples, and 27,181 samples, which corresponds to 0.1%, 0.5%, 1%, and 1.3% of all possible sample locations, respectively. The first hidden ANN layer consistently has 128 neurons. The plot labeled *Training time* in Fig. 7 shows the time required to train the ANN. The plot labeled *Training error* shows the final MSE of the ANN; ideally, this would be 0% (i.e., the output label obtained precisely matches the expected label for



Fig. 7 Increasing ANN training data typically increases ANN accuracy, but large training data sets make ANN training difficult at the expense of increased training time

training samples). The plot labeled *Testing error* shows the MSE for 2,244 additional randomly-selected samples that are not used for training, which shows the ANN’s accuracy on not-yet seen samples: these samples are taken from 0.1% of all possible samples on circuits not used for training (see Table 2).

Figure 7 shows clear trends regarding ANN training time and error. Training requires more time when using more data; this is because minor changes in ANN weights and biases are more likely to create error among more training data samples. Likewise, using more training data

can increase the error, since finding ANN weights that satisfy all training samples becomes more difficult.

Since using 20,910 samples doubled training time with a marginal impact on error, future experiments will use ANNs trained with 10,457 training data samples.

5.2 ANN Complexity

Many hyperparameters impact ANN complexity, but this study simplifies complexity to a single variable: the number of neurons present in the first hidden layer. Like the previous experiment, Fig. 8 gives a plot of ANN training time, training error, and testing error. 10,457 data samples train the stuck-at fault targeting, control-1 TP-evaluating ANN with all other parameters matching the previous experiment.

As with the previous experiment, Fig. 8 shows that more neurons in the first layer translates to more time needed to minimize ANN error, but training error and testing error also decrease. This is because using training with more neurons can find more accurate correlations between features and desired labels, but at the same time, training with more neurons requires more time to learn these correlations.

6 ANN vs. Heuristic TPI Experiments

6.1 Experimental Setup

Industry-representative workstations performed this study’s fault simulation and TPI using original software. These workstations use Intel i7-8700 processors and possess 8 GBs

Table 2 TPI Experiments

Bench	Benchmark Information					TPI Time(s)			
	Inputs	Outputs	Gates	Test Vectors	TPs	[26]	ANN (L = 3)	ANN (L = 4)	ANN (L = 5)
c432	36	7	203	9984	2	13.89	0.08	0.06	0.18
c880	60	26	469	9984	4	45.45	0.08	0.07	0.8
c1908	33	25	938	9984	9	911.08	0.44	0.36	4.17
c3540	50	22	1741	9984	5	1831.23	0.66	0.53	4.39
c5315	178	123	2608	9984	6	1806.72	1.76	1.39	7.45
b03	34	34	190	9984	1	1.21	0.89	0.73	0.07
b05	35	70	1032	9984	10	727.15	1.20	1.01	5.38
b07	50	57	490	9984	4	56.16	8.76	7.14	0.86
b09	29	29	198	9984	1	2.04	5.55	4.63	0.07
b11	38	39	801	9984	3	119.14	4.37	3.73	1.16
b13	63	63	415	9984	4	9.74	7.60	12.46	0.63
b14	277	299	10343	960	1	5031.64	5.27	4.47	5.69
b15	485	519	9371	1920	1	6472.05	5.71	4.85	5.27
b17	1452	1512	33741	256	1	26607.69	11.57	9.73	15.65
b20	522	512	20716	448	1	14479.94	11.67	9.91	11.52
b21	522	512	21061	448	1	14972.79	19.80	15.87	11.86

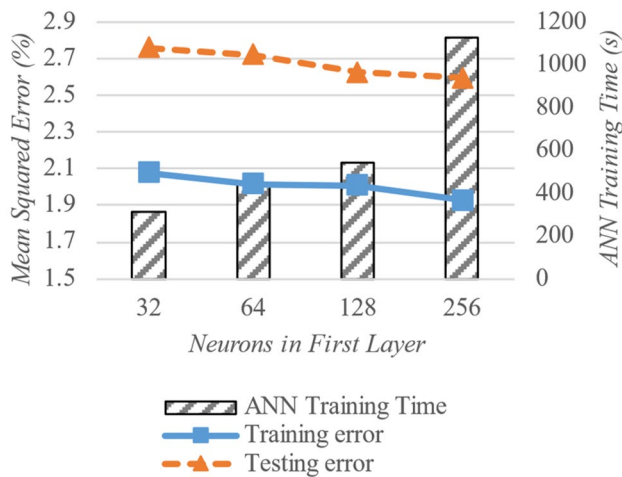


Fig. 8 Increasing ANN complexity can decrease ANN error, but because training time increases substantially, investments in complexity may not always be warranted

of RAM, and all software is implemented in C++ through the MSVC++ 14.15 compiler with maximum optimization parameters. This study uses original software in lieu of industry tools to obtain a fair comparison of the proposed ANNs against methods from literature: only the code which analyzes the TPs differs between the methods, thereby minimizing other sources of CPU time differences.

The goal of this study is to observe the impact of replacing critical subroutines with creatively-trained ANNs, thus in lieu of industrial-designed TPI tools with many difficult-to-isolate sources of CPU time, the conventional stuck-at fault TPI method used for comparison is from Tsai et al. [26]. Conveniently, this article's ANNs can directly replace the TP-evaluating subroutine in Tsai et al. [26], which eliminates other sources of fault coverage and CPU time differences. This heuristic is admittedly aged compared to contemporaries, like Maghaddam et al.'s [14], but the purpose of this study is not to propose a new TPI technique/algorithm itself. Instead, this study shows the efficacy of using ANNs in lieu of circuit-wide heuristics can have on a TPI time and quality, thus it is logical to presume applying this study's methods to the heuristics of more contemporary TPI – like Maghaddam et al.'s [14] – will yield similar trends in CPU time reduction and fault coverage improvement. Also, industrial tools will presumably be optimized to run on larger circuits, but trends seen in this study's experimental data do not appear to degrade as circuits get larger, thus this study's contributions will presumably apply to both large and small circuits.

Further modifications to Tsai et al. [26] implement a comparison delay fault-targeting TPI algorithm: the calculation proposed by Ghosh et al. [6] replaces Tsai et al.'s [26] TP-evaluating subroutine that estimates the detection probability

of a stuck-at fault (the controllability of a line multiplied by the observability of the line). Ghosh et al. [6] calculated the probability a TDF is observed as the controllability of a line, multiplied by the inverse of this controllability, multiplied by the observability of the line. Again, since the proposed ANN uses the same circuit information to evaluate the quality of a TP (i.e., COP values), the only difference in run-time and fault coverage quality is from this calculation.

ISCAS'85 [3] and ITC'99 [5] benchmark circuits not used for training are used for performing TPI; this prevents a bias favoring the ANN. Table 2 gives details of these benchmarks.

LBIST tests from a 64-bit PRPG generate different numbers of vectors for each benchmark circuit. These tests obtained 95% fault coverage without TPs, applied 10,000 vectors, or took at most 15 min to simulate. This represents an industrial environment where either (1) the circuit does not obtain 95% fault coverage without TPs and TPs are placed to increase fault coverage as much as possible, or (2) 10,000 vectors do not obtain 95% fault coverage and TPs attempt to increase fault coverage to acceptable levels. Since this study performs experiments in an academic setting, fault coverage results may be less compared to industrial environments, but industrial environments should still benefit from this study. For this study's original software, fault simulation may not obtain significant fault coverage in reasonable time (e.g., *b14* obtains a minuscule 13% fault coverage in 15 min when applying 960 vectors), but this study presumes high-performance industrial programming will remedy this and that fault coverage trends seen in this study will apply when simulating more vectors.

To explore the impact of ANN subcircuit sizes, this study trained several ANN models. For larger subcircuits, the number of neurons in the ANN increases to capture more complex relationships between input features and the output label. The number of neurons in each ANN model is 128 for $L = 2$, 256 for $L = 3$, and 512 for $L = 4$. All other hyperparameters are identical. With all these configurations, this means a total of $2 \cdot 3 \cdot 3 = 18$ neural networks were trained, which each being trained for a combination of fault model (stuck-at fault or transition delay fault), TP to analyze (control-0, control-1 and observe), and subcircuit size to analyze.

All TPI runs insert the same number of TPs, which is restricted to 1% of all gates or a 15-min TPI runtime. This limits larger circuits to a single TP, but industrial tools and contemporary TPI algorithms will presumably see similar TPI time and fault coverage trends when inserting more TPs.

6.2 Fault Coverage

6.2.1 Stuck-at Fault Coverage

These experiments find the stuck-at fault coverage of the four TPI methods: the stuck-at fault-targeting heuristic

(“SAF Heuristic”) [26], the ANN targeting stuck-at faults (see Sect. 3), the delay fault-targeting heuristic (“TDF Heuristic”) [6, 26], and the ANN targeting delay faults (see Sect. 3.3). Figures 9 and 10 plot the base stuck-at fault coverage (i.e., the original circuit with no TPs) and the change in stuck-at fault coverage after TPI.

Performing stuck-at fault simulation shows several noteworthy trends. First, all ANNs consistently obtain favorable stuck-at fault coverages, but the heuristic [26] sometimes (i.e., for *b05*) chooses TPs that decrease stuck-at fault coverage: this can occur because the heuristic chooses a control TP that (when active) may test for some new faults, but previously detected faults are no longer excited or prevented from propagating through the circuit [19]. Most ANNs select better TPs than their conventional counterparts, especially for larger ANNs that always outperform their conventional counterparts. Second, as the ANN analyzes larger subcircuits, it selects higher quality TPs and further increases stuck-at fault coverage, presumably because TP evaluation is more accurate. Third, as shown in Fig. 10, it appears that methods that target stuck-at faults (both heuristics and ANNs) are not consistently better at increasing stuck-at fault coverage compared to delay fault-targeting TPI. This implies that targeting only delay faults for TPI (and perhaps other test methods) can be more than sufficient to detect stuck-at faults, but it also begs the question of how to meet sufficient stuck-at fault coverage if delay fault coverage is not one hundred percent, which is typical in industry. This warrants future studies.

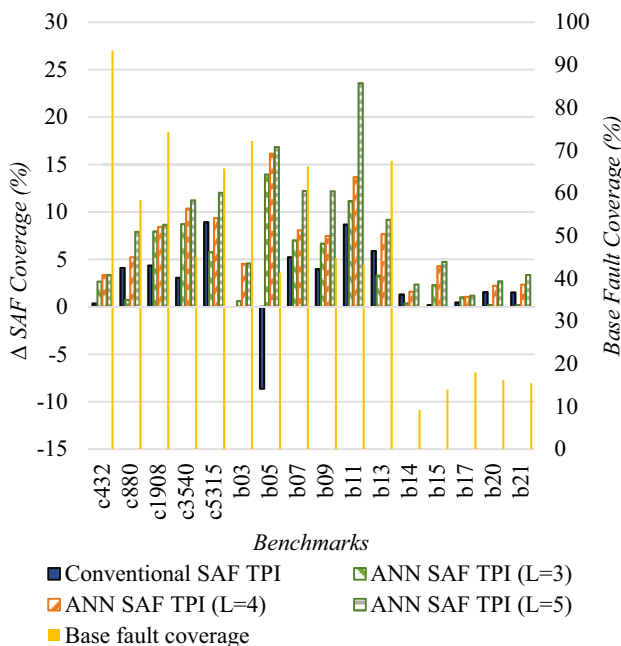


Fig. 9 As subcircuit sizes grow, the stuck-at fault-targeting ANNs improve stuck-at fault coverage more and better outperform its heuristic counterpart

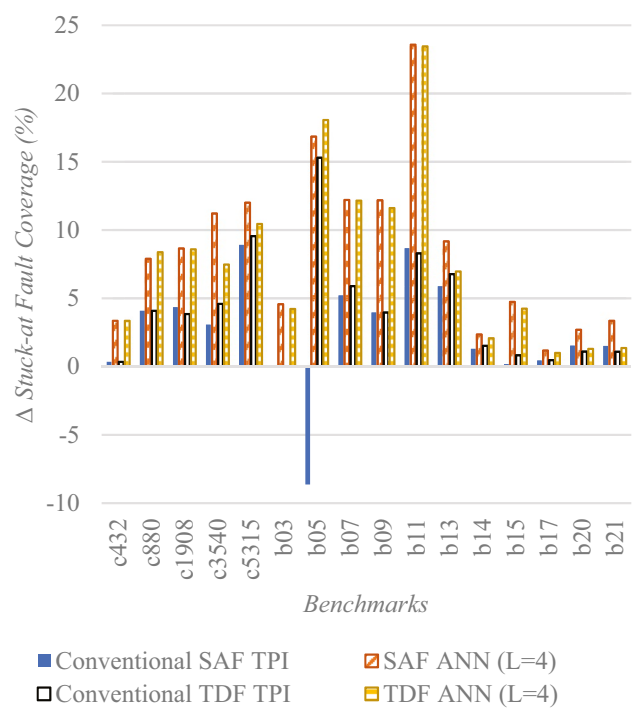


Fig. 10 Although counterintuitive, using delay fault-targeting TPI methods does not degrade stuck-at fault coverage

6.2.2 Delay Fault Coverage

Like the previous experiment, this experiment simulates delay faults (more specifically, TDFs). 64-bit PRPGs load the scan chains and apply vectors using a launch-off-scan method (a.k.a. “skew load” [20]). Although this was also done for the previous experiment, it was not relevant for stuck-at fault simulation. Figures 11 and 12 plot the delay fault coverage in terms of the base delay fault coverage (i.e., the transition fault coverage with no TPs) and the change in fault coverage with TPs inserted with the different TPI methods.

The first observation from Figs. 11 and 12 is that all ANN-based methods consistently obtain favorable delay fault coverage results. On average, ANNs select TPs that increase delay fault coverage more than heuristics do, regardless of which faults the ANN targets. Also like the stuck-at fault-targeting ANN, as the delay fault-targeting ANN analyzes larger subcircuits, the fault coverage is increased by selecting higher quality TPs.

From these results, it is clear that ANNs can effectively evaluate TPs for increased fault coverage, and the accuracy concerns posed in Sect. 3 were overcome. However, if finding these TPs requires substantial CPU time, then these results are moot, thus motivating the next section.

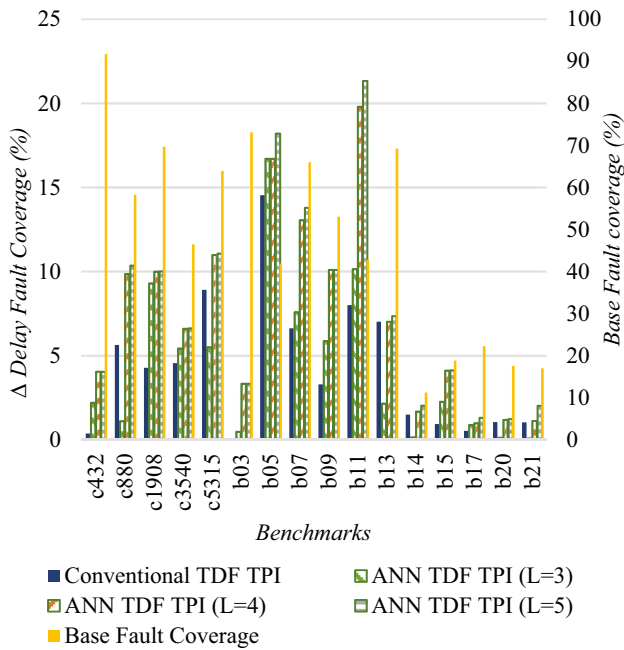


Fig. 11 As subcircuit sizes grow, the delay fault-targeting ANNs improve delay fault coverage more and ANNs further outperform heuristic counterparts

6.3 Time to Perform TPI

An additional result extracted from the previous experiments was the time required to perform TPI. This is given Fig. 13 in under the heading “TPI Time (s)” for the stuck-at fault-targeting

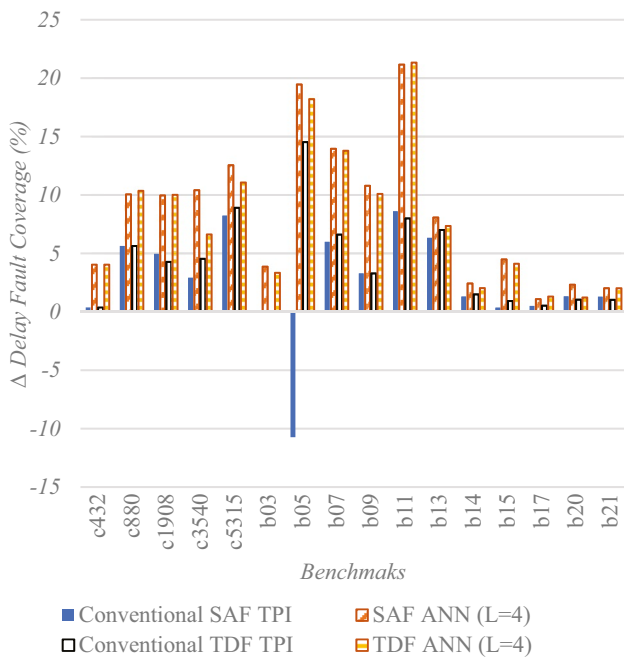


Fig. 12 Like stuck-at fault-targeting experiments, the delay fault-targeting heuristic and ANNs may not increase delay fault coverage compared to their stuck-at fault-targeting counterparts

heuristic (“Conventional TP”) and stuck-at fault-targeting ANNs with vector subcircuit sizes in Fig. 13. The times for other ANNs and heuristics are omitted for brevity, as trends are analogous. This figure has separate plots for the ANNs that include and exclude the training data generation and ANN training time; the plots with this training data generation and training time distribute this time among benchmarks by circuit size (i.e., by adding more time to circuits with more logic gates). ANNs with $L = 3$ (785 s), $L = 4$ (2,353 s), and $L = 5$ (5,753 s) have different times for generating training data and training the ANN.

The TPI time results from Table 2 and Fig. 13 definitively show that performing TPI using an ANN requires **orders of magnitude** less time than conventional methods, which justifies motivations given in Sect. 2.1.1. Additionally, as the previous section showed, fault coverages obtained with these ANNs are comparable or superior to heuristic-derived results, which means the decreased TPI time does not sacrifice TP quality. Beyond this orders-of-magnitude decrease in runtime, the variations in runtime between the ANNs are noteworthy: it is unpredictable which ANN will be the fastest for smaller circuits, and runtimes converge for larger circuits. This is likely due to other system processes (i.e., the operating system) dominating the ANN computation time, which implies the evaluation time between large and small ANNs is minimal, thus larger ANNs can be trained and used with minimal (if any significant) impact on CPU time. This implies industrial users should attempt to train the largest ANNs they can as long as they can be trained.

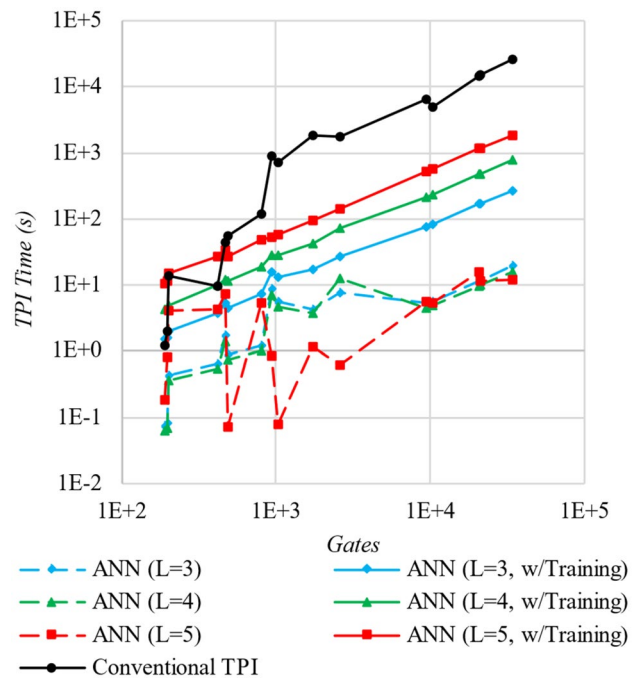


Fig. 13 Regardless of the ANN size and complexity, using ANNs significantly decreased TPI time, even when considering training time

When accounting for ANN training data generation and training time, the time to perform TPI still favors ANN-based TPI. The authors argue that not including this time is more representative of an industrial environment, i.e., when the EDA company trains the ANN and then circuit developers re-use the ANN numerous times. If developers re-use the tool enough times, the impact of training time becomes negligible. However, when distributing the training time among TPI instances, computational time still favors ANN-based TPI methods. This would represent a technique of training specifically for a single user; although this is impractical, this still out-performs heuristic TP evaluation.

7 Conclusions & Future Directions

Results showed that using ANNs to select TPs obtains better results compared to conventional heuristics in substantially less time. This article extended work done in previous studies [13, 22] by analyzing features relevant to LBIST tests (i.e., the number of vectors in the test), explored a more relevant label for evaluating a TP in the context of an LBIST test, and made a thorough exploration of the impact ANN complexity has on TPI outcomes. The results showed that by using ANNs to evaluate TPs, TPI time can be reduced by orders of magnitude without sacrificing LBIST quality and ANNs can be trained to evaluate TPs under more complex fault models (i.e., TDFs).

The authors anticipate continuing this work by finding features and training methods that further increase TP quality while inserting TPs under additional constraints, especially methods that consider test power while using nuanced delay fault models. This study chose features used by existing heuristics, thus making a fair comparison possible; however, additional features (e.g., circuit structure, critical paths, etc.) may drastically increase the ANN's quality and reveal nonintuitive causes of fault coverage degradation. Additionally, leveraging high-performance computing can further increase the ANN's complexity and performance. Adding constraints to TPI (e.g., area, power, and delay) will require such methods, which the authors believe is critical to applying this work to industrial circuits. The authors are also exploring extending the proposed method to other fault models, like path delay faults and cell-aware testing.

Applying ANNs to DFT challenges is a relatively new field of study, and the authors will continue exploring the benefits ANNs can bring to these challenges. Many DFT problems are still based on heuristics that do not scale favorably with increasing circuit sizes, which includes weighted random pattern selection, *automatic test pattern generation* (ATPG), fault diagnosis, and many more. If studies apply ANNs to these problems, one can foresee streamlined circuit development with significantly reduced

development costs. However, the end point of such returns should also be found: at what point will ANNs no longer reduce CPU time or increase algorithm outcomes, and at that point, what other computing techniques should the EDA community explore? The authors are most interested in exploring this last question and hope their next work will give the answer.

Data Availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflicts of Interests The authors have no relevant financial or non-financial interests to disclose.

References

1. Acero C, Feltham D, Liu Y, Moghaddam E, Mukherjee N, Patyra M, Rajski J, Reddy SM, Tyszer J, Zawada J (2017) Embedded Deterministic Test Points. *IEEE Trans Very Large Scale Integr VLSI Syst* 25(10):2949–2961
2. Brglez F (1984) On Testability Analysis of Combinational Networks. In: *Proc IEEE Intl Symp Circuits and Systems (ISCAS)*, Montreal, Quebec, Canada, May 1984, vol 1, pp 221–225
3. Brglez F, Fujiwara H (1985) A Neutral Netlist of 10 Combinational Benchmark Circuits and a Targeted Translator in FORTRAN. In: *Proc. IEEE Intl. Symp. Circuits and Systems (ISCAS)*, Kyoto, Japan, pp 663–698
4. Chern M et al (2019) Improving Scan Chain Diagnostic Accuracy Using Multi-stage Artificial Neural Networks. In: *Proc. 24th Asia and South Pacific Design Automation Conf.*, Tokyo, Japan, Jan 2019, pp 341–346
5. Davidson S (1999) ITC'99 Benchmark Circuits - Preliminary Results. In: *Proc Intl Test Conf (ITC)*, Atlantic City, NJ, USA, Sep. 1999, p 1125
6. Ghosh S, Bhunia S, Raychowdhury A, Roy K (2006) A Novel Delay Fault Testing Methodology Using Low-Overhead Built-In Delay Sensor. *IEEE Trans Comput Aided Des Integr Circuits Syst* 25(12):2934–2943
7. Gómez LR, Wunderlich H (2016) A Neural-Network-Based Fault Classifier. In: *Proc. 25th IEEE Asian Test Symposium (ATS)*, Hiroshima, Japan, Nov. 2016, pp 144–149
8. Haykin SO (2008) *Neural Networks and Learning Machines*, 3rd edn. Pearson, New York
9. Karystinos GN, Pados DA (2000) On Overfitting, Generalization, and Randomly Expanded Training Sets. *IEEE Trans Neural Netw* 11(5):1050–1057
10. Kingma DP, Ba J (2019) Adam: A Method for Stochastic Optimization. Dec. 2014. <http://arxiv.org/abs/1412.6980>. Accessed 8 Apr 2019
11. Ma Y, Ren H, Khailany B, Sikka H, Luo L, Natarajan K, Yu B (2019) High Performance Graph Convolutional Networks with Applications in Testability Analysis. In: *Proc. 56th Annual Design Automation Conference (DAC)*, New York, NY, USA, Article no. 18, pp 1–6
12. Menon A, Mehrotra K, Mohan CK, Ranka S (1996) Characterization of a Class of Sigmoid Functions with Applications to Neural Networks. *Neural Netw* 9(5):819–835

13. Millican SK, Sun Y, Roy S, Agrawal VD (2019) Applying Neural Networks to Delay Fault Testing: Test Point Insertion and Random Circuit Training. In: Proc. 28th Asian Test Symp. (ATS), Kolkata, India, pp 13–18
14. Moghaddam E, Mukherjee N, Rajski J, Solecki J, Tyszer J, Zawada J (2019) Logic BIST with Capture-per-Clock Hybrid Test Points. *IEEE Trans Comput Aided Des Integr Circuits Syst* 38(6):1028–1041
15. Nag PK, Gattiker A, Wei S, Blanton RD, Maly W (2002) Modeling the Economics of Testing: a DFT Perspective. *IEEE Des Test Comput* 19(1):29–41
16. Nigh P, Gattiker A (2000) Test Method Evaluation Experiments and Data. In: Proc. International Test Conference (ITC), Atlantic City, NJ, USA, pp 454–463
17. Pomeranz I (2017) Computation of Seeds for LFSR-Based n-Detection Test Generation. *ACM Trans Des Autom Electron Syst* 22(2):1–13
18. Rajski J, Tyszer J (1998) Arithmetic Built-in Self-test for Embedded Systems. Prentice-Hall Inc, Upper Saddle River, NJ, USA
19. Roy S, Stiene B, Millican SK, Agrawal VD (2019) Improved Random Pattern Delay Fault Coverage Using Inversion Test Points. In: Proc. IEEE 28th North Atlantic Test Workshop (NATW), Burlington, VT, USA, pp 1–6
20. Savir J (1992) Skewed-Load Transition Test: Part I, Calculus. In: Proc. Intl. Test Conf. (ITC), Baltimore, MD, USA, Sep. 1992, pp 705–713
21. Seshan K (2018) Reliability Issues: Reliability Imposed Limits to Scaling. In: Seshan K, Schepis D (eds) Handbook of Thin Film Deposition, 4th edn. William Andrew Publishing, pp 43–62
22. Sun Y, Millican SK (2019) Test Point Insertion Using Artificial Neural Networks. In: Proc. IEEE Computer Society Annu. Symp. VLSI (ISVLSI), Miami, FL, USA, pp 253–258
23. Sun Y, Millican SK, Agrawal VD (2020) Special Session: Survey of Test Point Insertion for Logic Built-in Self-test. In: Proc. IEEE VLSI Test Symposium (VTS), San Diego, CA, USA, pp 1–6
24. Sziray J (2011) Test Generation and Computational Complexity. In: Proc. IEEE 17th Pacific Rim Intl. Symp. Dependable Computing, Pasadena, CA, USA, pp 286–287
25. Tamarapalli N, Rajski J (1996) Constructive Multi-phase Test Point Insertion for Scan-based BIST. In: Proc Intl Test Conf (ITC), Washington, DC, USA, pp 649–658
26. Tsai H-C, Cheng K-T, Lin C-J, Bhawmik S (1997) A Hybrid Algorithm for Test Point Selection for Scan-based BIST. In: Proc. 34th ACM/IEEE Design Automation Conf. (DAC), Anaheim, CA, USA, pp 478–483
27. Xiang D, Wen X, Wang L-T (2017) Low-Power Scan-Based Built-In Self-Test Based on Weighted Pseudorandom Test Pattern Generation and Reseeding. *IEEE Trans Very Large Scale Integr VLSI Syst* 25(3):942–953

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Yang Sun received her B.S. degree in electrical engineering from the China University of Petroleum in 2016 and her M.S. and Ph.D. degrees from Auburn University in 2018 and 2021, respectively. She is currently a staff engineer at Marvell Technology in Santa Clara, CA. Her research interests include applying neural networks to design-for-test challenges.

Spencer K. Millican received his Ph.D., M.S., and B.S. degrees from the University of Wisconsin – Madison in 2015, 2013, and 2011, respectively. He was with IBM in Rochester, MN as a design-for-test engineer for two years and is currently an assistant professor at Auburn University. He has published several articles, including receiving the best paper award at the 2014 IEEE International Conference on VLSI Design, and he has received patents in the field of encrypted circuit simulation. His research interests include logic built-in self-test for modern technologies and encrypted circuit implementation and simulation.