Assessing Neural Networks as Guides for Testing Activities

Charles Anderson
Colorado State University
Computer Science Department
Fort Collins, CO 80523
anderson@cs.colostate.edu

Anneliese von Mayrhauser
Colorado State University
Computer Science Department
Fort Collins, CO 80523
avm@cs.colostate.edu

Tom Chen
Colorado State University
Electrical Eng. Department
Fort Collins, CO
tchen@cs.colostate.edu

Abstract

As test case automation increases, the volume of tests can become a problem. Further, it may not be immediately obvious whether the test generation tool generates effective test cases. Indeed, it might be useful to have a mechanism that is able to learn, based on past history, which test cases are likely to yield more failures versus those that are not likely to uncover any. We present experimental results on using a neural network for pruning a testcase set while preserving its effectiveness.

1 INTRODUCTION

Software testing still consumes major resources in a software product's lifecycle. While test data generation from some formal representation (syntactic or semantic) of the object under test can be very simple and cheap, forcing test criteria compliance is quite often expensive or only partially feasible. Examples are test generation tools based on symbolic execution that ensure various types of white box testing coverage [12]. This should come as no surprise, since the general test data generation problem is undecidable [7]. In practice, many test data generation systems make simplifying assumptions, either as to the power of the language for which they generate test data [2, 6, 20], or as to which information they consider in driving test data generation (e.g., Robert Poston's T tool). Effects of such simplification include: (1) the generated test data may not completely satisfy a test criterion, (2) the test data set may be larger than necessary, (3) the test data generation system may only be able to generate test data for a subset of the programming language. Simplification is obviously not a problem when test data are effective (reveal faults). This makes it very important to evaluate a test generation method experimentally and/or analytically. Analytical evaluation considers whether the testing criteria meet adequacy axioms [11, 17]. They describe properties any testing criteria should have to be considered adequate. They do not, however, guarantee high failure yields. Thus experimental evaluation of testing techniques is also important. Examples of such evaluations include [18, 19, 5].

When the number of tests to be run can become very large, test case reduction becomes desirable. Examples of reducing the number of test cases are [2, 16]. Prudent test case reduction prunes a test set by eliminating those test cases that are not likely to yield any failures.

This paper concentrates on experimental effectiveness analysis and prediction for two test scenarios. The first scenario analyzes cost-effectiveness predictions by a neural network for system tests generated using *Sleuth* [13] and application domain based testing [15]. We used a neural network as a classifier to learn about the system under test and to predict the fault exposure capability of newly generated test cases. We describe attributes of test cases to the neural network as inputs and relate them to resulting faults (neural network outputs). Inputs are test case length and various coverage metrics relevant to the testing strategy used. Outputs are levels of severity of faults detected. They range from severe (level 1) to "correct operation" (level 4). Then, we trained the network to recognize relationships between test case descriptors and faults. Once trained, the network acts as a fault predictor for new test cases. Related work includes the use of neural networks to predict quality in a manufacturing process [1].

The second scenario evaluates how well neural networks predict test coverage of VHDL models[1]. Given a method of producing plausible tests, can a neural network predict which tests are most likely to produce the desired coverage? Since a neural network can predict coverage orders of magnitude faster than actual simulation, this can save time by reducing the number of simulations, assuming the prediction is sufficiently accurate. If this is indeed the case, we can use the neural net to

- filter out (possibly automatically generated test cases) that are not likely to increase coverage and thus will not add to the fault revealing capabilities of an existing test suite, reducing the test effectiveness of the test set, and
- reduce the already existing set by throwing out "coverage duplicates" and thus keep test cases to a minimum while still achieving coverage goals.

Can the trained neural network be used to generate new test cases that result in new coverage not produced by existing test cases?

Section 2 provides background on neural network classifiers and their use to predict failures. Section 3 introduces the experimental design to evaluate how well neural networks work when used as test effectiveness predictors. As example testing technique we use Domain Based Testing [13] and its associated test data generation tool *SLEUTH* [15]. The results are useful in two ways. First, they point out which of the generated test suites are likely to trigger what types of incidents. Second, for those test suites that the neural net has identified as not revealing any problems, the neural net analysis can also provide information about what test variables to change so as to produce effective results. This is the first step towards prescribing successful test generation, where success is defined as finding faults. Section 4 describes the experiment with the VHDL model. The analysis uses a realistic VHDL design. Section 5 summarizes our conclusions.

2 BACKGROUND ON NEURAL NETWORKS

2.1 CLASSIFIERS

Neural network methods were developed to model the neural architecture and computation of the human brain [8]. A neural network consists of simple *neuron-like* processing elements. Processing elements are interconnected by a network of weighted connections that encode network knowledge. Neural networks are highly parallel and exercise distributed control. They emphasize automatic learning. Neural networks have been used as memories, pattern recall devices, pattern classifiers, and general function mapping engines [3, 8, 21]. Test effectiveness evaluation concentrates on their use as pattern classifiers. A classifier maps input vectors to output vectors in two phases. The network learns the input-output classification from a set of training vectors. After training, the network acts as a classifier for new vectors.

Figure 1 shows the anatomy of a processing element (also called a node, unit, processing unit, or neuron). Node output or activation, $o(\vec{x}, \vec{w})$, is a function of the weighted sum (dot product) of the input vector $\vec{x}$ and the interconnection weights $\vec{w}$. A common activation (output) function is the unipolar sigmoid with a range from 0.0 to 1.0. The activation equations are

$$\begin{aligned} sum &= \sum_{i=0}^{n} x_i w_i \\ o(sum) &= \frac{1}{1 + e^{-sum}} \end{aligned}$$

In our work we used a neural network defined by the number of nodes in the input layer (input units), the hidden layer (hidden units), and the output layer (output units) (see Figure 2(a)). The

[1] If we consider VHDL as similar to a programming language, then results are applicable to other languages as well

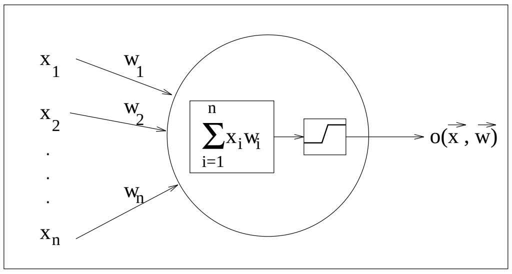


Figure 1: Anatomy of a Neuron

number of input units and the number of output units are problem dependent. The number of hidden units is usually not known. Hidden units allow the neural network to learn by forcing the network to develop its own internal representation of the input space. The network that produces the best classification with the fewest units is selected as the best topology. A net with too few hidden units cannot learn the mapping to the required accuracy. Too many hidden units allow the net to "memorize" the training data and does not generalize well to new data.

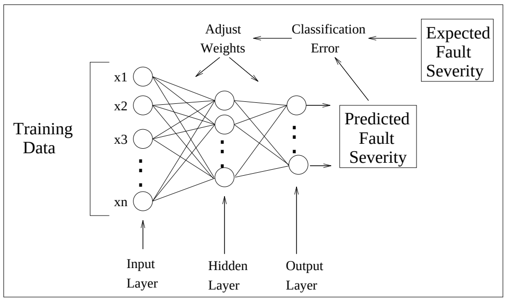
Backpropagation is the most popular training algorithm for multilayer neural networks [3, 8, 21]. The algorithm initializes the network with a random set of weights, and the network trains from a set of input-output pairs. Each pair requires a two-stage learning algorithm: forward pass and backward pass. The forward pass propagates the input vector through the network until it reaches the output layer (see Figure 2(a)). First, the input vector propagates to the hidden units. Each hidden unit calculates the weighted sum of the input vector and its interconnection weights. Each hidden unit uses the weighted sum to calculate its activation. Next, hidden unit activation propagates to the output layer. Each node in the output layer calculates its weighted sum and activation. The output of the network is compared to the expected output of the input-output pair, and their difference defines the classification error. In the second stage of network training, the output error propagates backward to update network weights. First, the error passes from the output layer to the hidden layer updating output weights. Next, each hidden unit calculates an error based on the error from each output unit. The error from the hidden units updates input weights. Details about the Backpropagation algorithm can be found in [3, 8, 21]. One training *epoch* passes when the network sees all input-output pairs in the training set. Training stops when the sum squared error is acceptable or when a predefined number of epochs passes.

Once trained, network weights are fixed and the net acts as a pattern classifier (see Figure 2(b)). As a classifier, the network examines input vectors it has never seen and predicts the class of the input vector as explained in Section 3.

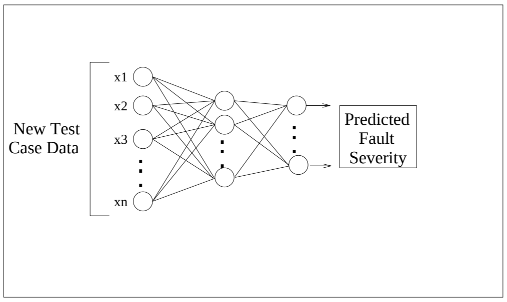
2.2 FAULT PREDICTION

Once a software test generator creates a test case, we would like to evaluate its fault exposure potential. If the test case does not expose faults, then we don't need to run it. We plan to use a neural network classifier to make this decision. Figure 3 shows how to train the neural network. Test case metrics are extracted from the test case. Metrics measure test case length, command frequencies, and parameter use frequencies. An "oracle" classifies any errors exposed by the test case. The oracle is an objective arbiter to whether a given test case exposes a fault. In practice, testers act as arbiters, but in our controlled laboratory experiment we defined a synthetic test oracle.

The network uses test case metrics (input pattern) and the error classification (output pattern) for training. Once trained, the network predicts the fault exposure of new test cases. Figure 4 shows how we evaluate the effectiveness of the neural classifier. Given a test case, the test oracle identifies the actual faults exposed by the test case and the neural network predicts the fault exposure. Comparing the two, we measure how well the neural net acts as a test case effectiveness predictor.



a.



b.

Figure 2: Training Phase and Prediction Phase of the Multilayer Neural Network

3 EXPERIMENT DESIGN - SYSTEM TESTING

We conducted an empirical study to show the effectiveness of a neural net fault predictor. The study used the DBT test generation tool *Sleuth* to generate test data [15, 13]. Using test case metrics, a synthetic test oracle evaluated each test case for error classification. The neural net trained on test metric input patterns and mapped them to the test oracle's error classification. Once trained, the network acts as a test case effectiveness predictor.

3.1 TEST DATA GENERATION

The empirical study needed a data set to train and evaluate the neural net. Using the *Sleuth* test generator, we defined six test objectives from which 180 data vectors were generated. Each test objective is associated with a set of test criteria. For the experiment, we trained and tested the neural network on a set of commands commonly used to control an automated tape library. This includes commands for mounting and dismounting tapes, and for moving tapes between tape silos. Domain

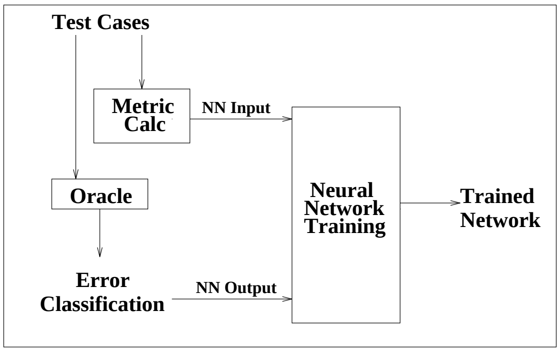


Figure 3: Neural Net Test Effectiveness Predictor : Training Phase

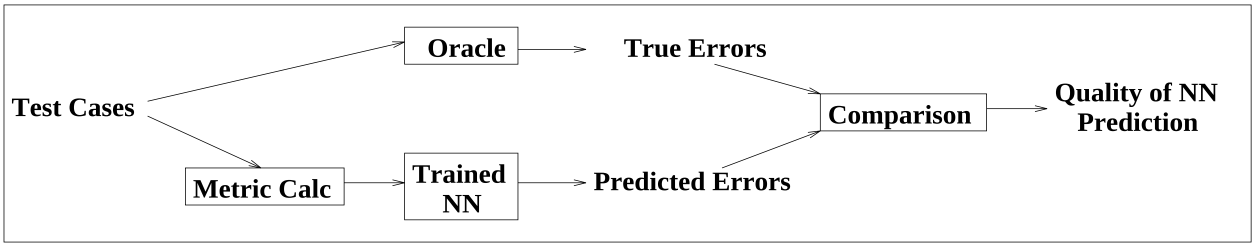


Figure 4: Neural Net Test Effectiveness Predictor : Evaluation Phase

Based Testing defines formal models a applications under test that vary with the test objective. The test suite generation tool, *SLEUTH* uses these models (also called testing subdomains) as the basis from which to generate test suites. Such models can also include rules for proper sequencing of commands, and rules for parameter dependencies in the language. In our example, tapes must have been mounted before they can be dismounted, tapes cannot be moved to slots where they already reside, etc. Applications must have a command language user interface.

Table 1 lists six test objectives used to generate training and test data. We also show test oracles the test case is likely to invoke (see next section for test oracle descriptions). The first test objective, D_0^{NN}, is to test the application for valid operation (mounting and dismounting tapes, moving tapes around, etc.). All rules representing sensible application use are to be observed. This test objective represents testing normal or valid system operation. Systematic feature testing must go beyond testing of "normal" behavior and test invalid or pathological use of the software. Test objectives 2-6 reflect this by systematically disregarding various types of rules of sensible system use. The second test objective TSD_{NoMD}^{NN} disregards the `Mount-Dismount` scripting rules. The robot tape library requires `Mount` and `Dismount` commands to be sequenced properly. It does not make sense to issue a `Dismount` request unless a tape was `Mount`ed earlier in the command sequence. Test objective TSD_{NoED}^{NN} tests for faults in the `Enter-Drain` sequences. The `Enter` command allows a test engineer to insert new tapes into the robot tape library through a door called a Cartridge Access Port. The door is a shared resource and it is assigned to one tester at a time. When finished, the tester issues a `Drain` command to release the door for the next tester. Test objective four, TSD_{NoSR}^{NN}, disregards all scripting rules. In this case, the test generator creates random lists of commands, but within commands, relationships between parameter values are still enforced during the parameter value selection for test commands. The next test objective, TSD_{NoICR}^{NN} explicitly disregards such intracommand rules. Intracommand rules specify how to choose parameter values within a single command. Sometimes the value of one parameter constrains the choices of other parameters in the same command. For instance, when moving tapes within the same "silo", it is usually not allowed to move tapes to the very slots they just came from (meaning that in a move command the slot identifier parameter where the tape is now and where it should be moved, must be different). The last test objective, $TSD_{NoRules}^{NN}$, is to disregard any and all rules. This effectively generates random sequences of commands and parameter values. Thus, the six test objectives represent a variety of sensible and pathological system uses.

3.2 TEST ORACLE

The test oracle acts as an impartial, objective arbiter to determine whether a given test case (i.e., sequence of commands) exposes a fault and if so, the type of fault. The oracle for this study was synthetic in that it did not model actual software behavior. It provides a controlled laboratory environment to analyze the neural network results. When used in the field, the test oracle is replaced by testers who judge the results of running a test case.

Table 2 specifies characteristics of ten hypothetical faults in an automated tape library, representing six different aspects of the application domain. These ten hypothetical faults cause failures at three

	Test Objectives	Possible Faults (Oracles)
1	Normal system use (D_0^{NN})	1,4-7,10
2	Violate Mount-Dismount Rules (TSD_{NoMD}^{NN})	2,4-7,10
3	Violate Enter-Drain Rules (TSD_{NoED}^{NN})	1,3,4-7,10
4	Random commands (TSD_{NoSR}^{NN})	2-7,10
5	Violate parameter relationships (TSD_{NoICR}^{NN})	1,4-10
6	Random command and parameter values ($TSD_{NoRules}^{NN}$)	2-10

Table 1: Test Objectives in Neural Network Prediction Experiments

different severity levels. Severity 1 is the most severe and Severity 3 the least. Severity 4 represents no-fault. For each fault, Table 2 gives the symptom and the specification how to recognize it (fault indicator column), as well as the severity of the problem. The first three faults focus on improper command sequences. Oracle #1 models a situation with correct `Mount-Dismount` behavior, but a low severity fault because of a slight error in the mount scheduler for large requests. The second oracle identifies a condition where there are more `Dismount` requests than `Mount` requests, representing faulty error handling on the part of the tape library software. The third problem identifies an anomaly when a user does not release the CAP correctly.

The chronic command fault identifies problems in the database that maintains tape locations. Here, long test sequences and too many `Disable` requests cause database inconsistencies. The next two faults focus on a chronic command fault coupled with a particular parameter value. If the `MODIFY` command is issued to turn the tape silo LSM 000 `Online`, it will not work properly. If the frequency of the request is less than three, then it takes the system an unusually long time to change the LSM's state (Severity 3). If more than three requests are issued, then the silo stays `Offline` and the fault is more severe (Severity 2). The seventh fault focuses on the Pass Through Port (PTP) between two silos. If we have a long test case and there are more than eight PTP `Move` requests, then a tape can be lost. Faults eight and nine represent incorrect error handling for violations of intracommand rules. For a short test case, the system fails to issue a warning message for violating the rule. If the test is long, the tape is moved to the wrong destination panel. The last fault occurs with interacting commands. If the ratio of `Mount` an d `Move` commands is greater than 0.8, the CDS becomes inconsistent. This means the system cannot handle significantly more tape moves than tape mounts. Table 2 also shows the formal specification of the fault indicator for each fault.

3.3 NEURAL NETWORK TRAINING

We trained four neural nets using error backpropagation with unipolar sigmoid units. Each network learned to predict one fault severity level. We used 21 input nodes and one output node for each network. The number of hidden units was calculated experimentally to achieve the best Root Mean Square (RMS) error during training. Network training started with random initial weights between -0.005 and +0.005. Before training, all input and output vectors in the data set were normalized using a linear scale[2]. The minimum value for each input parameter is set to 0.0 and the maximum value is

[2]Scaling reduces the side effects of scale differences between parameters. Linear, square root, logarithm, and general data transformation are typical scaling methods [9].

set to 1.0. Output parameters are set to 0.1 for a "zero" response and 0.9 for a "one" response.

The test data set for neural net training included 180 observations, thirty test cases for each of the six test objectives. Table 3 shows an example test case generated from TSD^{NN}_{NoED}. Table 4 shows the 21 metrics used as input to the neural net and lists the four output severity levels. These inputs relate to coverage metrics for Domain Based Testing, i.e., we wanted to explore with the fault revealing capabilities of the testing technique itself, to which extent coverage measures of a test case work in training a neural network to recognize effective test cases. The first input calculates the length of the test case, the next ten inputs represent the frequency of each command in the test case, and the last ten inputs count the number of unique values for each parameter. The output vector identifies fault severity levels 1-4. Table 4 also shows a data vector for the example test case in Figure 3. As indicated, two faults are exposed by this test case. The first identifies a Severity 2 fault when the `ENTER` command is not followed by a `DRAIN` command. The second fault identifies problems with the "`MODIFY 000 Online`" command.

We trained all four networks (one for each severity level) using the Leave-One-Out Method (LOOM). LOOM is most useful for relatively small data sets like ours. We used LOOM to experimentally calculate the best network topology (number of hidden units). Table 5 lists the best topology for each network. Fault Severity 1 through 3 needed a single hidden unit, while Severity 4 used five.

3.4 EXPERIMENTAL RESULTS

The results from training all four networks are listed in Table 6. The second column shows how well each network predicted individual fault severities. The Severity 4 (no error) predicted the best with 94.4%, and the second-best predictor was the Severity 1 (most severe fault) net, with 91.7%. We placed the incorrectly classified tests into one of three categories: *False Positive*, *False Negative*, and *Other Incorrect*. A False Positive response is recorded for severity 1-3 errors when the network predicts a fault that doesn't truly exist. For S4 (no error), a false positive means that the neural network predicted that there was not an error, when indeed the test case would have uncovered one. For severity classes 1-3, a False Negative response is recorded when the neural net predicted Severity 4 (no fault exposed) when the test case indicates a fault. Other Incorrect refers to tests that were classified by the neural net as exposing a fault, but of the wrong type. We use this information to analyze three test data generation objectives.

Objective 1: Reduce Number of Test Cases. One goal for test data generation is to reduce the number of test cases run on the system under test. Each test case consumes machine time and resources. In addition, testers must evaluate each test case to determine whether a fault was exposed. To reduce testing time and cost, we need to run tests that are likely to identify a fault. This is particularly important with automated test data generation systems that easily and quickly generate thousands of test cases.

A tester could use the neural net classifier to reduce the number of test cases. The neural net classifies each test case. Severity 4 predictions need not be run. Two things that help us achieve this objective are automated test data generation and low cost of test case evaluation. Automated test data generators like *Sleuth* generate test cases quickly. They can generate tests much more quickly than it takes to run them. Likewise, a neural network predictor evaluates a test case in a single forward-pass through the net. The key to meeting this objective is a predictor that is good at the binary decision: `Does this test case expose a fault or not`? We aren't concerned about misclassification of the individual fault severity but we must keep False Positive predictions for severity levels 1-3 to a minimum. It is also advantageous to keep False Positive low for Severity 4.

Our study shows that the neural net has a low False Positive and Negative prediction rates. This suggests that the neural net can be used to reduce the number of test cases. We also note that with these results we could reduce the number of test cases to be run by 30%.

Objective 2: Emphasize Severe Error Exposure. Another goal for test data generation is to create tests that expose the most severe system faults. The cost to fix such problems once the product is in the field is high. To make matters worse, severe faults are often the most difficult to isolate, identify, and expose.

A tester could meet this test data generation objective using a neural net to evaluate tests before they are run. If the net predicts high severity (e.g., Severity 1 or 2 for our tests), then the test should be run. We also have the advantage of automated test data generation and a fast test evaluation. The key to achieving this objective is an accurate fault prediction at the higher levels of severity. We are concerned about misclassification of tests at lower severity levels. If the net tends to classify test cases with a lower fault severity than truly exists, then tests will not run that should. If misclassified with higher fault severity than truly exists, then we will run tests that need not run. We also require the neural net to have a low False Negative prediction. If tests that are likely to expose faults are classified as Severity 4, we will not run them, but we should.

Data from the empirical study suggests that the neural network can be used to identify tests for severe error exposure. The network predicts Severity 1 and Severity 2 faults well, and its False Negative rate is low. Only 12 out of 180 vectors were False Negative. We did not find any tests in which the network predicted a higher severity fault than really existed.

Objective 3: Minimize Number of Test Cases. Another test data generation goal is to minimize the total number of test cases to run. This can be viewed as a combination of the first two objectives. First, we need to reduce the number of tests run and we also need to accurately predict the fault exposure at each severity level. We keep test cases that expose errors at multiple severity levels.

One way to rank a test case as more effective than another is to count the number of fault severity levels the test exposes. If one test exposes multiple severity levels (say, Severity 2 and 3), it should be ranked higher than a test that exposes a single level (i.e., Severity 2). A test case that exposes multiple severity levels meets several test criteria, it reduces the number of test cases to run, and it *reduces* the total test set for the system under test. To achieve this objective, we need accurate test case effectiveness prediction. We emphasize the accuracy of the neural net prediction because we want to keep those tests that identify multiple severity levels. If the net misclassified a multi-severity test as a single severity test, then it may not be ranked high enough to be included in the test set.

The data set for the experimental study contains 60 multi-severity vectors. The neural net fails to predict every fault in 26 multi-severity test vectors. When failing to predict every fault, these 26 vectors could be ranked incorrectly and some of them could be eliminated from the minimal test set. The variability in these classifications can be explained by comparing the test oracle with the test case metrics used by the network. The test oracle has "complete" information to identify faults through the error specification of Table 2. The neural network only gets summarized information via the metrics in Table 4. The network sees a "filtered" picture of each test case. So it is not surprising that it does not predict multi-severity level tests accurately. What these results indicate, however, is that the neural network produces remarkably good "coarse" level predictions (i.e., Test Data Objectives 1 and 2) despite the limited metric information. In particular, the network effectively screens no-yield test cases (Test Objective 1) and identifies high severity tests (Test Objective 2). This result also indicates that we need to add other test case metrics to improve neural net predictions.

4 EXPERIMENT DESIGN - VHDL CODE COVERAGE

4.1 VHDL Model and Branch Coverage Data

For the experiment we chose a VHDL model of a VLSI chip used for visual object recognition [10]. This model has about 3,700 lines of VHDL code. The code is of the same character and complexity as is found in microprocessors and thus represents a realistic problem. It presents a challenging design validation task, because it consists largely of random logic. Manual generation of a validation suite would require understanding the 3,700 lines of VHDL and finding a set of stimuli which produce the desired branch coverage in the model.

VLSI design validation is usually done by simulating the device on a computer, using a variety of manually and automatically generated stimuli. The automatically generated stimuli often consist of random bits placed in a template to exercise various parts of the design. The simulation results are then checked by hand, or automatically with the use of an oracle—a program which can determine whether results are correct.

To investigate automated generation of stimuli for behavioral designs, we randomly generated 400 independent tests, ran them through the simulator, and recorded the results for later training and analysis by a neural network. Each test run consisted of the following steps:

1. The simulator was reset, and the branch coverage flags set to FALSE.
2. The clock line (of the simulated VLSI device) was set high and the simulation time advanced a few nanoseconds in order to allow the device to respond and stabilize.
3. The 36 inputs (not including the clock line) were set to a random state (each bit randomly set to 0 or 1 with 50% probability), and the state of each of these input bits was recorded in a file. The simulation time was advanced again.
4. The clock line was set low, and time was advanced.
5. Steps 3 and 4 are repeated two more times.

This model generated 505 coverage flags; 450 of them were constant over all test runs, with 148 of the branches always covered and 302 of them never covered. This is not surprising, as some branches will always be covered by any clock activity, and some branches will never be covered with just 1 1/2 clock cycles. The mean coverage over all 400 patterns is 169.8 with a standard deviation of 4.6. The range is from 155 to 182. The constant branch flags were eliminated from the collected data, leaving 55 bits of coverage data for use in training and analysis of the neural net. Thus, each test produced 36 x 3, or 108, bits representing the stimulus to the device, and 55 bits representing the coverage results from that stimulus.

4.2 NEURAL NETWORK TRAINING AND RESULTS

A number of different neural network architectures were tested. Performance of each was measured by the percent of branch coverage flags correctly predicted as being covered or not. A network output greater than 0.5 is taken as a prediction that the corresponding branch would be covered.

Figure 5 shows the percent correct versus the network architectures. Architectures are indicated by a pair of numbersgiving the number of hidden units in each layer. A zero for the second number represents a network with a single hidden layer. Plotted values are averages over 20 training runs, each with different initial weight values and different selections of training, validation and test data.

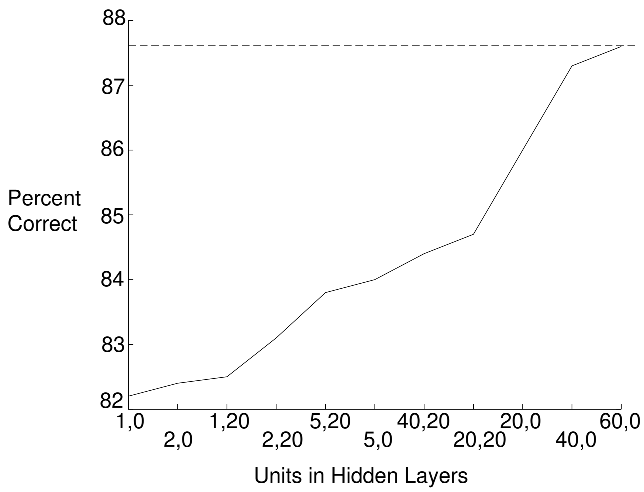


Figure 5: Fraction of coverage flags correctly predicted, averaged over 20 runs.

The figure shows that even a network with a single hidden unit can, on average, correctly predict 82% of the branch coverage flags. A network with a single hidden unit is practically a linear function.

The addition of up to 60 hidden units, however, improves performance to almost 88% correct. Larger networks might result in even higher accuracy, but this has not been tested.

A count of correct and incorrect predictions made by a 40,0 network are shown in Table 7. The test data consists of 80 cases of 55 branch flags each, for a total of 4,400 flags to be predicted. The table shows that out of these 4,400, 1,558 of the covered branches and 2,394 of the uncovered branches were correctly predicted, making a total of 3,952, or 89.8%, correct predictions. The table also shows that the network produced 248 false positives (predicted coverage for uncovered branches) and 200 false negatives (no predicted coverage for covered branches).

Thus, the neural network prediction method is able to predict coverage with fairly high accuracy. Now we describe some preliminary results from inverting the trained network to generate new test stimuli.

A random input stimulus was supplied to the neural network. This input stimulus was incrementally modified by performing a gradient-based minimization search, where the gradient is of the squared difference between the network's output and the desired coverage. The desired coverage was set to either all branches covered or all branches uncovered. This process was repeated 5 times with a desired coverage of all branches, and 4 times with a desired coverage of no branches. The resulting input stimuli were then passed through the VHDL simulator and the resulting coverage recorded. The generated input stimuli consist of real-valued components. To pass the generated input stimuli through the simulator, these values are converted to binary values by setting them to 1 if greater than 0.5 and to 0 if not.

Table 8 summarizes the results. Input stimulus 1 is predicted by the network to cover 181 branches. When applied to the VHDL simulator, 185 branches were covered. Recall that the largest number of branches covered by any single test in our initial 400 stimuli is 182, so this procedure has discovered a new test stimulus that covers more branches than any previous single stimulus. This table also shows that those test stimuli generated to maximize coverage do indeed cover more branches than those stimuli generated to minimize coverage.

This result relates to individual test stimuli, but how well does the coverage produced by the set of 5 generated test stimuli compare to that of the initial 400 stimuli? The initial set of 400 stimuli covers 203 branches. The set of generated stimuli, numbered 1–5 in Table 8 covers 196 (96.6%) of these while 7 remain uncovered. Inspection of exactly which branches are covered by each test stimulus reveals that the 5 generated test stimuli do not cover any additional branches that were not covered by the initial set of 400 stimuli.

5 CONCLUSIONS

A neural network has been shown to be a useful approach to test case effectiveness prediction. The neural net formalizes and objectively evaluates some of the testing folklore and rules-of-thumb that are system specific and often require many years of testing experience. A neural network is not system nor test case metric specific. Therefore, it can be used across a variety of testing criteria, test case and coverage metrics, and fault severity levels.

To continue this work, we need additional experiments. First, we must train and evaluate neural network predictions using actual test data. This involves two parallel efforts. We need to identify a set of test case metrics that are likely indicators for fault exposure. The metrics used for the experiment were loosely based on coverage metrics for Domain Based Testing. In general, the set of metrics needed for an arbitrary testing criterion is not known. One approach would be to identify a wide range of metrics, train the neural net, and reduce the set of metrics to the most likely fault indicators. A field study would identify fault exposure on actual test cases as judged by test engineers.

Test case metrics that indicate fault exposures are likely to change as software matures. We also know that neural networks are adaptive. Through continued training or retraining, a neural net can modify its predictions. We need to experiment with the adaptive nature of a neural net. Using a synthetic oracle in a controlled experiment, we can remove some faults and add new ones. Responses from the corresponding oracle could train a neural network or retrain an old one. After the controlled experiments with the test oracle, the adaptive nature of the neural net needs to be evaluated with field data.

Based on our experiment, neural networks are promising test case effectiveness predictors. The networks are system independent and are able to adapt as the software matures. The accuracy of the neural net appears to be sufficient to predict test case effectiveness, and we can trust the predictions to meet test data generation objectives.

The results for increasing test coverage (in our case of a VHDL model) are not quite as encouraging. While the neural net learned coverage and reliably reduces the test set, it does not generate test cases that increase coverage. We believe that this could be remedied by including more structural information about the code to make it successful at increasing coverage.

Acknowledgements

This research was partially supported by the Colorado Advanced Software Institute (CASI) and StorageTek. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

References

[1] Charles W. Anderson, Judy A. Franklin, and Richard S. Sutton. "Learning a Nonlinear Model of a Manufacturing Process Using Multilayer Connectionist Networks," *Proc. of the 5th IEEE International Symposium on Intelligent Control*, Philadelphia, PA, September 1990, pp. 404–409.

[2] DeMillo, R., A.; Offutt, A., J.; "Constraint-Based Automatic Test Data Generation", *IEEE Transactions on Software Engineering SE-17*, 9(Sept. 1991), pp. 900–910.

[3] Laurene Fausett. *Fundamentals of Neural Networks.* Prentice Hall: Englewood Cliffs, New Jersey, 1994.

[4] R. Hamlet, R. Taylor; "Partition Testing does not inspire confidence", *IEEE Transactions on Software Engineering vol. 16*, no. 12(Dec. 1990), pp. 1402–1411.

[5] P. Frankl, S. Weiss; "An experimental comparison of the effectiveness of branch testing and data flow testing", *Transactions on Software Engineering vol. 19*, no. 8(August 1993), pp. 774–787.

[6] Higashino, T., v. Bochman, G.; "Automatic Analysis and Test Case Derivation for a Restricted Class of LOTOS Expressions with Data Parameters", *IEEE Transactions on Software Engineering SE-20*, 1(January 1994), pp. 29–42.

[7] W. Howden; *Functional Testing and Analysis*,
McGraw-Hill, New York, NY, 1987.

[8] J.L. McClelland, D.E. Rumelhart, and the PDP Research Group. *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, vol 1. MIT Press, 1986.

[9] Robert G.D. Steel and James H. Torrie. *Principles and Procedures of Statistics: A Biometrical Approach*, second edition. McGraw-Hill: New York, 1980.

[10] Maureen Schaffer and Tom Chen. A vlsi architecture for 2d object classification based on tree matching. In *Proceedings of the International Conference on Computer Architectures for Machine Perception*, September 1995.

[11] A. Parrish, S. Zweben; "Analysis and Refinement of Software Test Data Adequacy Properties", IEEE Transactions on Software Engineering vol. 17, no. 6(June 1991), pp. 565–581.

[12] Gmeiner, L.; Voges, U.; von Mayrhauser, A.; "SADAT–An Automated Testing Tool", *IEEE Transactions on Software Engineering SE-6*, 3(May 1980), pp. 286–290.

[13] Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz, "Testing Applications Using Domain Based Testing and *Sleuth*," *Proceedings of the Fifth International Software Reliability Engineering Symposium*, Monterey, November 1994, pp. 206–215.

[14] Anneliese von Mayrhauser, Richard Mraz, Jeff Walls, and Pete Ocken. "Domain Based Testing: Increasing Test Case Reuse," *Proc. of the International Conference on Computer Design*, Boston, October 1994, pp. 484–491.

[15] Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz. "*Sleuth*: A Domain Based Testing Tool", *Proc. of the International Test Conference*, October, 1994.

[16] von Mayrhauser, A.; Anderson, Ch.; Mraz, R.; "Using A Neural Network to Predict Test Case Effectiveness", *Procs. IEEE Aerospace Applications Conference*, Snowmass, CO, Feb. 1995

[17] E. Weyuker; "Axiomatizing Software Test Data Adequacy", *IEEE Transactions on Software Engineering vol. 12*, no. 12(Dec. 1986), pp. 1128–1138.

[18] E. Weyuker; "The Cost of Data Flow Testing: An Empirical Study", *IEEE Transactions on Software Engineering vol. 16*, no. 2(Feb. 1990), pp. 121–128.

[19] E. Weyuker; "More Experience with Data Flow Testing", *IEEE Transactions on Software Engineering vol. 19*, no. 9(Sep. 1993), pp. 912–919.

[20] Weyuker, E., Goradia, T., Singh, A.; "Automatically Generating Test Data from a Boolean Specification", *IEEE Transactions on Software Engineering SE-20*, 5(May 1994), pp. 353–363.

[21] Jacek M. Zurada. *Introduction to Artificial Neural Systems.* West Publishing: St. Paul, 1992.

Domain Model Focus		Symptom	Fault Indicators	Sev
Sequencing Fault	1	Incorrect mount order	Test Case Length ≥ 30 MOUNT Freq = DISMOUNT Freq	S3
	2	Incorrect command sequencing	Test Case Length ≥ 20 MOUNT Freq < DISMOUNT Freq	S2
	3	CAP isn't released	Test Case Length ≥ 10 ENTER Freq > DRAIN Freq	S2
Chronic Command Fault	4	Inconsistent database	Test Case Length > 3 0 CDS Freq ≥ 5 `disable` Parameter > 3	S1
Chronic Command & Parameter Fault	5	Unusually high delay to put 000 online	`MODIFY 000 ONLINE` MODIFY Frequency ≤ 3	S3
	6	System stays offline	`MODIFY 000 ONLINE` MODIFY Frequency > 3	S2
Object Fault (Pass Through Port)	7	Lost tape	Test Case Length > 30 Number of PTP Moves > 8	S1
Intracommand Rule Fault	8	No warning issued for violating the rule.	Test Case Length ≤ 15 LSM1 = LSM2 AND Panel1 = Panel2	S3
	9	Tape is moved to wrong destination panel	Test Case Length > 15 LSM1 = LSM2 AND Panel1 = Panel2	S2
Command Interaction	10	Inconsistent database	Ratio: MOUNT Freq to MOVE Freq > 0.8 Test Case Length ≥ 30	S1

Table 2: Test Oracle Specification

MOVE (EVT393) Tlsm (000)
DISPLAY VOLSER EVT199
CDS Disable STandby
MODIFY 000 OFFline
ENTER 001
DISPLAY CAP
MODIFY 001 ONline
MODIFY 000 ONline
DISPLAY LSM 000
MOUNT EVT180 A17
EJECT EVT180 001
EJECT EVT185 010
MOVE (EVT199) Tlsm (000)
DISPLAY CAP
DISMOUNT EVT180 A17
SRVLEV BASE
DISPLAY SRVLEV

Table 3: Test Case Generated from TSD^{NN}_{NoED} for the Neural Network Experiments

Input Vector

Index	Description	Example
1.	Test Case Length	17
2.	CDS Frequency	1
3.	DISMOUNT Frequency	1
4.	DISPLAY Frequency	5
5.	DRAIN Frequency	0
6.	EJECT Frequency	2
7.	ENTER Frequency	1
8.	MODIFY Frequency	3
9.	MOUNT Frequency	1
10.	MOVE Frequency	2
11.	SRVLEV Frequency	1
12.	acs Frequency	0
13.	cap Frequency	2
14.	cc Frequency	0
15.	drive Frequency	1
16.	dsn Frequency	0
17.	host Frequency	0
18.	lsm Frequency	2
19.	pp Frequency	0
20.	rr Frequency	0
21.	volser Frequency	4

Output Vector

Index	Description	Example
22.	Severity 1 Indicator	0
23.	Severity 2 Indicator	1 (oracle 3)
24.	Severity 3 Indicator	1 (oracle 5)
25.	Severity 4 Indicator	0

Table 4: Input and Output Vector Descriptions

Severity	Input Units	Hidden Units	Output Units
S1	21	1	1
S2	21	1	1
S3	21	1	1
S4	21	5	1

Table 5: Network Topologies

Sev.	Correct Class	False +	False −	Other Incorrect
S1	165 (91.7%)	0	0	15
S2	161 (89.4%)	7	0	12
S3	149 (82.8%)	18	12	1
S4	170 (94.4%)	3	0	7

Table 6: Neural Network Prediction Results

		Actual Data	
		Covered	Not Covered
Network Prediction	Covered	1,558	200
	Not Covered	248	2,394

Table 7: Results for a 40,0 network.

	Input Stimuli	Branches Covered Predicted	Branches Covered Actual
Maximized Coverage	1	181	185
	2	181	182
	3	179	178
	4	178	181
	5	183	176
Minimized Coverage	6	156	155
	7	156	155
	8	157	156
	9	156	161

Table 8: Predicted and Actual Number of Branches Covered.