International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Impact Factor (2012): 3.358

Performance Based Evaluation of New Software Testing Using Artificial Neural Network

Jogi John[1], Mangesh Wanjari[2]

[1]Priyadarshini College of Engineering, Nagpur, Maharashtra, India

[2]Shri Ramdeobaba College of Engineering & Management, Katol, Nagpur, Maharashtra, India

Abstract: ***Today, testing is the most challenging and dominating activity used by industry, therefore, improvement in its effectiveness, both with respect to the time and resources, is taken as a major factor by many researchers. Software testing forms an integral part of the software development life cycle. Since the objective of testing is to ensure the conformity of an application to its specification, a test "Automated Secure Agent" is needed to determine whether a given test case exposes a fault or not. Using an automated Agent to support the activities of human testers can reduce the actual cost of the testing process and the related maintenance costs. In this paper, we present a new concept of using an artificial neural network as an automated agent for a tested software system. A neural network is trained by the back propagation algorithm on a set of test cases applied to the original version of the system. The network training is based on the "black-box" approach, since only inputs and outputs of the system are presented to the algorithm. The trained network can be used as an artificial Agent for evaluating the correctness of the output produced by new and possibly faulty versions of the software. We present experimental results of using a two-layer neural network to detect faults within mutated code of a small credit approval application. The results appear to be promising for a wide range of injected faults.***

Keywords: Test Data**,** Software Testing, ANN, Black-Box, White-Box, Regression Test, Automated Secure Agent.

1. Introduction

Testing software is essential to ensure quality in IT systems. The main objective of software testing is to determine how well an evaluated application conforms to its specification. Two common approaches to software testing are black-box and white-box testing. While the white-box approach uses the actual code of the tested program to perform its analysis, the black-box approach checks the program output against the input without taking into account its inner Workings. [1]Software testing is divided into three stages: generation of test data, application of the data to the software being tested, and evaluation of the results. Traditionally, software testing was done manually by a human tester who chose the test cases and analyzed the results.[2] However, due to the increase in the number and size of the programs being tested in present day, the burden of the human tester is increased, and alternative, automated software testing methods are needed. While automated methods appear to take over the role of the human tester, the issues of reliability and the capabilities of the software testing methods still need to be resolved.[3] Thus, testing is an important aspect in the design of a software product. Both the white-box and black-box approaches to software testing are not without their limitations. Voas and McGraw [1] noted that present-day software systems are too large to be tested by the white-box approach as a single entity; instead, white-box testing techniques work at the subsystem level. One of the limitations of the white-box testing approach is that it is not capable of analyzing certain faults, one of which is testing for missing code.[4] The main problem associated with the black-box approach is to generate test cases that are more likely to detect faults.[4] "Fault-based testing" is the term used to refer to methods that base the selection of test data on the detection of specific faults,[4] and is a type of white-box approach as it uses the code of the tested program.[1] Mutation analysis is a fault-based technique that generates mutant versions of the program that is being tested.[5]A test set is applied to every mutant program and is evaluated to determine whether the test set is able to distinguish between the original and mutant versions.

2. Background Work

Artificial neural networks (ANNs) have been used in the past to handle several aspects of software testing. Experiments have been conducted to evaluate the effectiveness of generating test cases capable of exposing faults, [6] to use principle components analysis to find faults in a system, [7] to compare the capabilities of neural networks to other fault-exposing techniques, [8], [9] and to find faults in failure data. [10] In this synopsis, new application of neural networks as an "automated secure agent" for a tested system is presented. A multi-layer neural network is trained on the original software application by using randomly generated test data that conform to the specification. The neural network can be trained within a reasonable accuracy of the original program, though it may be unable to classify the test data 100 percent correctly. In effect, the trained neural network becomes a simulated model of the software application. When new versions of the original application are created and "regression testing" is required, the tested code is executed on the test data to yield outputs that are compared with those of the neural network. Here it is assumed that the new versions do not change the existing functions, which means that the application is supposed to produce the same output for the same inputs. A comparison tool then makes the decision whether the output of the tested application is incorrect or correct based on the network activation functions. Figure 1 presents the overview of the proposed testing methodology for security check. Using an ANN-based model of the software, rather than running the original version of the program, may be advantageous for a variety of reasons. First, the original

version may become unusable, due to a change in the hardware platform or the OS environment. Another usability problem may be associated with a third-party application having an expired license or other restrictions. Second, most inputs and outputs of the original application may be non-critical at a given stage of the testing process, and, thus, using a neural network for an automated modeling of the original application may secure a significant amount of computer resources. Third, saving an exhaustive set of test cases with the outputs of the original version may be infeasible for real-world applications. [1] Finally, the original version is never guaranteed to be fault-free, and comparing its output to the output of a new version may overlook the cases where both versions do not function properly. Neural networks provide an additional parameter associated with every output, the activation function, which, as we show below, can be used to evaluate the reliability of the tested output.

3. Proposed Model

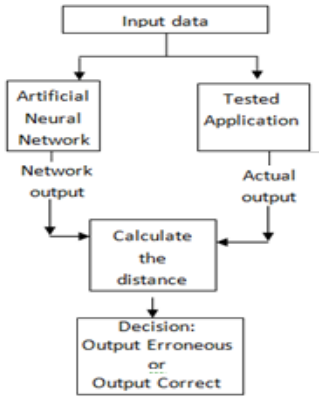


Figure 1: Overview of the Evolution Phase

The comparison tool is employed as an independent method of comparing the results from the neural network and the results of the tested versions of the credit approval application. An objective automated approach is required to ensure that the results have not been affected by external factors. This in effect replaces the human tester, who may be biased by having prior knowledge of the original application.

Table 1: Each output has a defined category.

	Tested application output	
ANN output	Correct	Wrong
	1	2
Correct	True positive	True negative
	4	3
Wrong	False positive	False negative

The tool uses the output of a neural network and the output of the tested application. The distance between the outputs is taken as the absolute difference between the value of the winning node for each output and the corresponding value in the application. Since a sigmoid activation function is used to provide the network outputs, the activation value of the winning output nodes is a number between 0.0 and 1.0. The corresponding value of the application output is equal to 1.0 if the predicted and actual outputs are identical. Otherwise, it is equal to 0.0. Thus, the distance covers a range between 0.0 and 1.0, and we use this value to determine whether the faulty application has generated an invalid or correct result.

Table I displays the four possible categories where each output can be placed. Since the ANN is only an approximation of the actual system, some of its outputs may be incorrect. On the other hand, the tested application itself may produce errors, which is the main reason for the testing process. If the ANN output is correct while the output of the tested application is wrong, the evaluation of the comparison tool is classified as being a true negative or a category of 2, i.e., the determination that the output of the application is an actual error. Similarly, the remaining three classifications represent the other possibilities for the output categorization. Each output arising from the neural network and the tested program is evaluated in this fashion. Although, the main interest is in finding the wrong outputs (categories 2 and 3), there is also no visible difference when the network output is the same as the output of the tested program (categories 1 and 3). Categories 2 and 4 are also similar in that regard, as either the network output is correct or the tested program output is correct, with the former being more likely. The ANN is trained to simulate the original application; however it is not capable of classifying the original data 100% correctly due to the problem of error convergence. Thus, consider only cases where the tested application output is wrong: categories 2 and 3, using the notation of Table I. When the outputs are compared with one another, they are either the same or different. Consequently, categories 1 and 3 have to be distinguished from one another by the comparison tool; a similar separation is required for categories 2 and 4. Thus, the need for calculating the distance is justified.

4. Design of Credit Card Approval Program

The sample program that is being tested in this experiment is a small credit approval application. The application can be considered representative of a wide range of business applications, where a few critical outputs depend on a large number of inputs. The training data that are used throughout this paper are randomly generated using the specification of the application and the description of the attributes. A more detailed description and the type of each attribute can be viewed in Table II, and Table III provides a sample data set.

Table 2: Input attributes of the data

Name of the attribute	*Data type*	*Attribute type*	*Details*
Serial ID	integer	Input	unique for each customer
Citizenship	integer	Input	0: American 1: Others
State	integer	Input	0: Florida 1: other states
Region	integer	Input	0–6 for different regions in U.S.
Income class	integer	Input	0 if income p.a. < $10k 1 if income p.a. ≥ $10k 2 if income p.a. ≥ $25k 3 if income p.a. ≥ $50k
Sex	integer	Input	0: Female 1: Male
Age	integer	Input	1–100
Number of dependents	integer	Input	0–4
Marital status	integer	Input	0: Single 1: Married
Credit amount	integer	Output	≥ 0
Credit approved	integer	Output	0: No 1: Yes

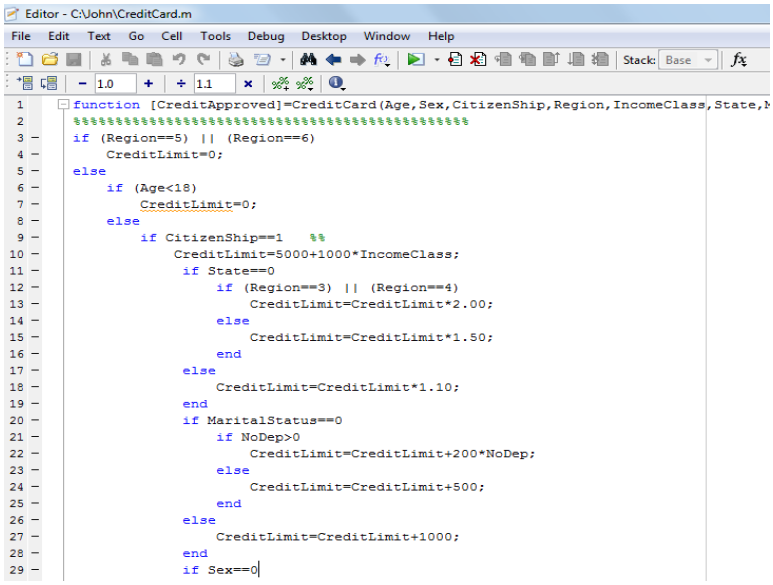
Table 3: Sample data used during training (before preprocessing)

Serial ID Number	*Citizenship*	*State*	*Region*	*Income class*	*Sex*	*Age*	*Number of Dependents*	*Marital status*	*Amount*	*Credit approved*
1	0	1	3	1	1	20	1	1	860	0
2	1	1	4	1	1	18	1	0	1200	0
3	0	0	5	1	0	15	0	0	0	1
4	0	0	3	1	1	53	0	1	1400	0
5	0	0	4	2	1	6	2	0	0	1
6	1	1	3	0	1	95	1	0	400	0
7	1	0	5	2	1	78	2	0	0	1
8	0	0	2	0	0	84	2	0	1650	0
9	0	1	3	2	0	28	3	1	1370	0
10	0	0	2	2	0	74	2	0	1950	0

For example, customer 2 of Table III is not an American citizen, does not live in Florida, is 18 years of age, is male, lives in region 4, has an annual income greater than $10,000, and is single with one dependent. Credit has been approved for this client for an amount of $1,200. Since a neural network can be trained only on numeric values, all categorical attributes (citizenship, state, and so on) were converted to numeric form. The training data consist of 500 records (test cases); the additional 1,000 test cases used for evaluating the mutated versions of the original application also follow the same format. The second data set is larger than the first to ensure that there were sufficient data to find faults in the tested program.

A detailed description of the application logic is necessary for the reader to understand the type of faults that are injected into the application though this logic was "hidden" from the back propagation training algorithm. The algorithm that the application follows can be found in Figure 1. The structure of the application consists of a series of layered conditional statements. This provides the opportunity to examine the effects of the faults over a range of possibilities. The types of faults that have been injected into our experiment consist of minor changes to the conditional statements. These include a change in operator and a change in the values used in the conditional statements. Several assumptions are made when applying the faults to the application. Only one change is made at a time, and the fault is either a sign change or an error in the numerical value used in the comparison. Consequently, the analysis of the outputs was conducted independently of each other.

5. Credit Card Application Algorithm



```
                    CreditLimit=CreditLimit+500;
                else
                    CreditLimit=CreditLimit+1000;
                end
            end
        end
%else
    CreditLimit=1000+800*IncomeClass;
    if MaritalStatus==0
        if NoDep>2
            CreditLimit=CreditLimit+100*NoDep;
        else
            CreditLimit=CreditLimit+100;
        end
        CreditLimit=CreditLimit+300;
    end
    if Sex==0
        CreditLimit=CreditLimit+100;
    else
        CreditLimit=CreditLimit+200;
    end
end
if CreditLimit==0
    CreditApproved=1
else
    CreditApproved=0
end
%CreditLimit
```

6. Back Propagation Algorithm for Testing

```
Editor - C:\John\ANN_BackPropogation.m
File Edit Text Go Cell Tools Debug Desktop Window Help

%%%% Program for ANN Training using Back Propogation FeedForward Algorithm
%%%% Author : Mr. John
%%%%% Date  : 21/02/2014
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x=xlsread('data.xlsx');
TrData=x(:,2:9);
TrData(:,9)=x(:,11);
P=TrData';
T=x(:,10)';
net=newff(P,T,[9 9 9 9],{'tansig','tansig','tansig','tansig'},'trainlm','learngdm','mse');
net.trainParam.epochs = 100;
net = train(net,P,T);
Y = sim(net,P);
err=abs(T-Y);
clc
disp('---------------------------------------------------------------------')
fprintf('\n .....BackPropogation FF Neural Network is successfully trained!...\n')
fprintf('\n No. of Input Layers  = 1\n')
fprintf('\n No. of Hidden Layers = 2\n')
fprintf('\n No. of Output Layers = 1\n')
fprintf('\n No. of Neurons used in each of the layers = 9\n')
fprintf('\n MSE (Mean Sqaured Error = %f\n',mean(err));
disp('---------------------------------------------------------------------')
P1=P(:,1);
Y1 = sim(net,P1);
if Y1<=0.5
    CreditApproved=0
else
    CreditApproved=1
```

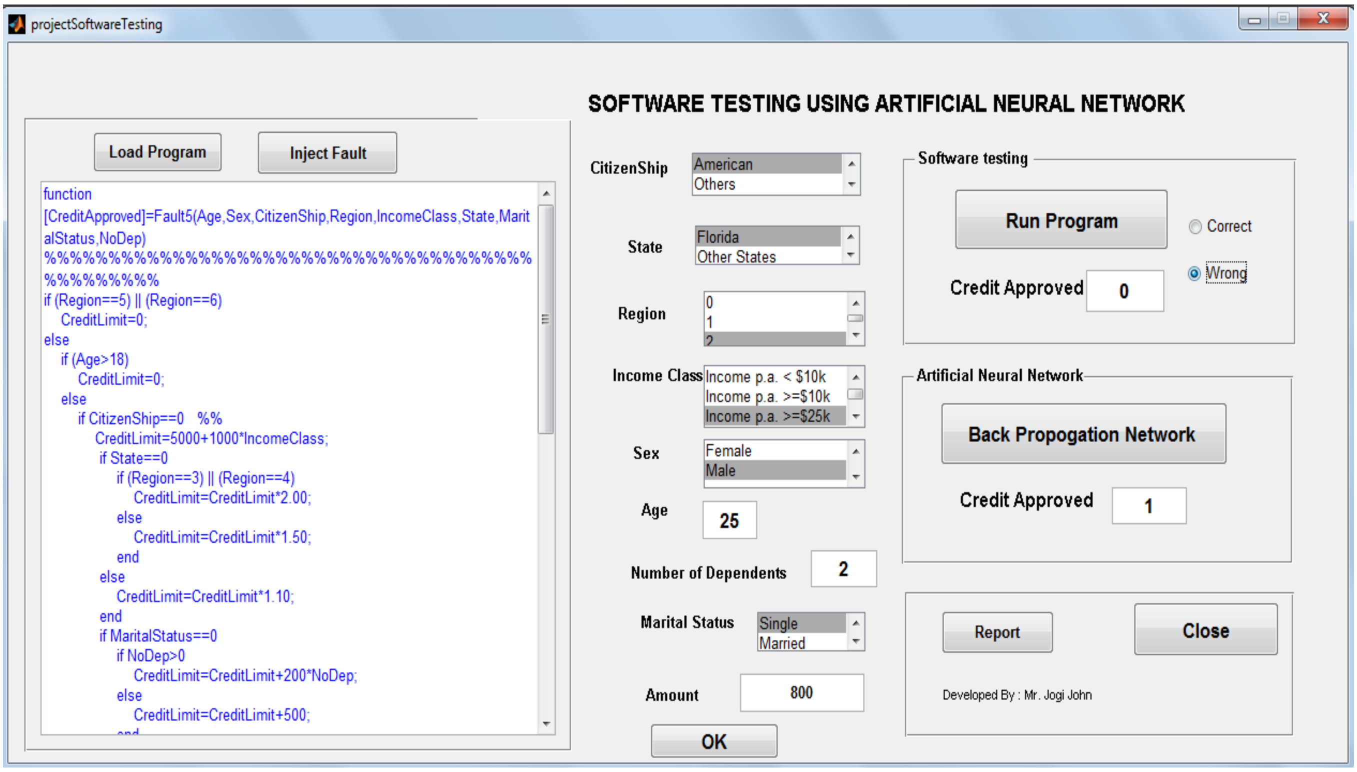


Figure 2: Proposed Model In MATLAB

7. Experiment Result

After loading the program as shown in the Fig 3 the faults are injected in the loaded credit card program. After injecting the faults the application program output is coming out to be wrong or correct. The ANN model is used to check the output after comparing the result it will listed out the faults which is been tested. Mean squared normalized error performance function by square rooting the ANN output square –program output square.

Table 1: List of Faults Tested

Fault#	Line#	Original Line	Injected Fault	Error type	Error	Confidence	MSE
1	1	if (Region=5) || (Region=6)	if (Region=5)	Operator Change & argument change	3.50	[2.90 4.10]	0.52
1	1	if (Region=5) || (Region=6)	if (Region=5)	Operator Change & argument change	3.50	[2.90 4.10]	0.52
2	1		if (Region=5) && (Region=6)	Operator Change	15.70	[14.5 16.90]	1.14
4	1		if (Region=3) || (Region=4	Argument change	40.70	[39.1 42.30]	2.28
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
8	9	if (State=0)	if (State=1)	Argument change	3.50	[2.90 4.10]	0.82
4	1		if (Region=3) || (Region=4	Argument change	40.70	[39.1 42.30]	2.28
5	4	if (Age < 18)	if (Age > 18)	Operator Change	80.70	[79.10 81.10]	4.12
6	4		if (Age < 25)	Argument change	9.20	[8.30 10.10]	0.52

The tables include the injected fault number, the number of correct outputs and incorrect outputs as determined by the "Automated Secure Agent" and the percentages for the

correct outputs classified as being incorrect and incorrect outputs

8. Conclusions

In this paper, we have used a neural network as an "automated Secure Agent" for testing a real application, and applied mutation testing to generate faulty versions of the original program. We then used a comparison tool to evaluate the correctness of the obtained results based on the absolute difference between the two outputs. The neural network is shown to be a promising method of testing a software application provided that the training data have a good coverage of the input range. The back propagation method of training the neural network is a relatively rigorous method capable of generalization, and one of its properties ensures that the network can be updated by learning new data. As the software that the network is trained to simulate is updated, so too can the trained neural network learn to classify the new data. Thus, the neural network is capable of learning new versions of evolving software. The benefits and limitations of the approach presented in this paper need to be fully studied on additional software systems involving a larger number of inputs and outputs. However, as most of the methodology introduced in this paper has been developed from other known techniques in artificial intelligence, it can be used as a solid basis for future experimentation. One possible application can include generation of test cases that are more likely to cause faults. The heuristic used by the comparison tool may be modified by using more than two thresholds or an overlap of thresholds by fuzzification. The method can be further evaluated by introducing more types of faults into a tested application.

References

[1] Voas JM, McGraw G. Software Fault Injection; 19985
[2] Choi J, Choi B. Test agent system design. In: 1999 IEEE International Fuzzy Systems Conference Proceedings; August 22–25, 1999.
[3] G. McGraw, "Building Secure Software: A Difficult but Critical Step in Protecting Your Business," Cigital, White Paper, available at: http://www.cigital.com/whitepapers/
[4] Weyuker E, Goradia T, Singh A. Automatically generating test data from a boolean specification.IEEE Transactions on Software Engineering 1994;SE-20(5):353–363.
[5] DeMillo RA, Offutt AJ. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 1991; SE-17(9):900–910.
[6] Anderson C, von Mayrhauser A, Mraz R. On the use of neural networks to guide software testing activities. In: Proceedings of ITC'95, the International Test Conference; October21–26, 1995.
[7] Khoshgoftaar TM, Szabo RM.Using neural networks to predict software faults during testing.IEEE Transactions on Reliability 1996; 45(3):456–462.
[8] Khoshgoftaar TM, Allen EB, Hudepohl JP, Aud SJ. Application of neural networks to software quality modeling of a very large telecommunications system. IEEE Transactions on Neural Networks 1997; 8(4):902–909.
[9] Kirkland LV, Wright RG. Using neural networks to solve testing problems. IEEE Aerospace and Electronics Systems Magazine 1997; 12(8):36–40.
[10] Sherer SA. Software fault prediction. Journal of Systems and Software 1995; 29(2):97–105.