Using a Neural Network in the Software Testing Process

Meenakshi Vanmali, Mark Last, Abraham Kandel

Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave., ENB 118
Tampa, FL 33620, USA

Please address all correspondence to:

Dr. Abraham Kandel, Chairman
Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave., ENB 118
Tampa, FL 33620, USA
Tel: +1 (813) 974-4232
Fax: +1 (813) 974-5456
e-mail: kandel@csee.usf.edu

ABSTRACT

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 “oracle” is needed to determine whether a given test case exposes a fault or not. Using an automated oracle 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 oracle for a tested software system. A neural network is trained by the backpropagation 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 oracle 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: Automated software testing, Mutation testing, Black-box testing, Artificial neural networks

1 INTRODUCTION

The main objective of software testing is to determine how well the evaluated application conforms to its specifications. 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 [10].

Software testing is divided into three stages: generation of test data, application of the data to the software being tested, and the evaluation of the results. Traditionally, software testing was done manually by a human tester who chooses the test cases and analyzes 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 capability of the software testing method still need to be resolved [11]. 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 & McGraw (1998) note 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 [12]. The main problem associated with the black-box approach is to generate test cases that are more likely to detect faults [12].

"Fault-based testing" is the term used to refer to methods that base the selection of test data on the detection of specific faults [12] and is a type of white-box approach as it is using the code of the tested program [10]. Mutation analysis is a fault-based technique that generates mutant versions of the program that is being tested [3]. 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.

Artificial neural networks (ANN) 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[1], to use principle components analysis to find faults in a system [6], to compare the capabilities of neural networks to other fault exposing techniques [5, 7], and to find faults in failure data [9].

In this paper, we present a new application of neural networks as an automated "oracle" for a tested system. A multi-layer neural network is trained on the original software application by using randomly generated test data that conform to the specifications. 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% 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. We assume here 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 1.1 presents the overview of the proposed testing methodology.

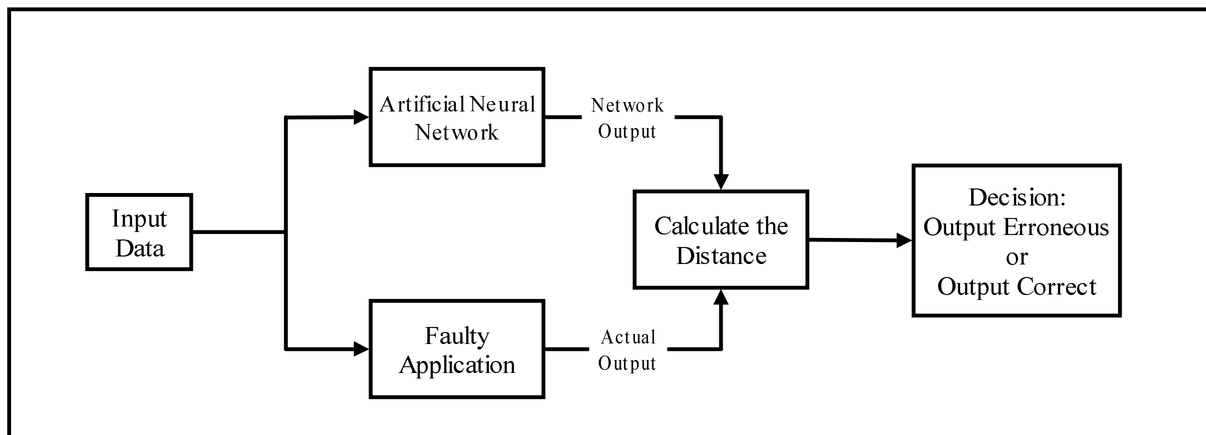

Figure 1.1 Method Overview

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 in 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 save 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 [10]. 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.

Section 2 describes the background material on neural networks and its effectiveness when used as a model of a real system. Section 3 presents a description of the proposed testing methodology. Section 4 describes the credit card approval system that is being used in our experiment as well as the construction and the training of the multi-layer neural network. Section 5 presents the results of the experiment

conducted to evaluate the effectiveness of using a neural network as an automated "oracle." Section 6 summarizes our conclusions.

2 BACKGROUND MATERIAL ON NEURAL NETWORKS

Artificial neural networks (ANNs) are designed to resemble the structure and information-processing capabilities of the brain [4]. The architectural components of a neural network are computational units similar to the neurons of the brain. A neural network is formed from one or more layers of these neurons whose interconnections have associated synaptic weight. Each neuron in the network is used to perform calculations that contribute to the overall learning process, or training, of the network. The neuron interconnections are associated with synaptic weights that store the information computed during the training of the network. The neural network is thus a massive parallel information processing system that utilizes distributed control to learn and store knowledge about its environment.

The two inherent factors that influence the superior computational ability of the neural network are its parallel distributed design and its capability to extrapolate the learned information to yield outputs for inputs not presented during training (generalization). These characteristics of the neural network allow complex problems to be solved. Data mining, pattern recognition, and function approximation are some of the tasks that are capable of being performed by neural networks.

In our method, the trained neural network is used to produce a particular output when presented with an input signal. During the training stage, the synaptic weights of the network are updated by presenting a set of training examples to the training algorithm. Each example is organized as a pair of two vectors, one for the inputs and the second for the outputs. One epoch is concluded when the entire set of examples is presented to the neural network. Thus, the neural network employs an input-output mapping mechanism and is expected to generalize when presented with new examples. One possible problem that may arise during the training of the network is that the network error may converge to a local minimum [4]. Retraining the network with a different set of initial weights is one solution to overcome this problem.

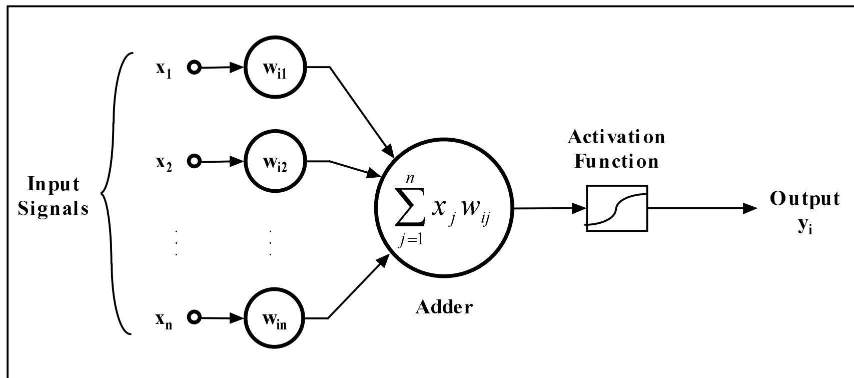

Figure 2.1 Structure of a Neuron

The computational components or neurons of the network are each composed of three elements: a set of *n* synaptic weights, one for each interconnection; an adder that will be used to combine the incoming input signals with the synaptic weights; and an activation function also known as a squashing function as it forces the output to have a value within a specified range. Figure 2.1 illustrates the structure of a neuron. If the input signal to neuron i is x_j, the synaptic weight associated with the interconnection

between the input signal and the neuron is denoted w_{ij}. The set of input signals multiplied with the set of the synaptic weights is linearly combined. The resulting value is then used to calculate the output of the activation function, typically a sigmoid function (shown in Figure 2.2) with a range between 0.0 and 1.0 that consequently generates the neuron output signal y_i.

$$net = \sum_{j=1}^{n} x_j w_{ij} \quad (1)$$

$$y_i = \frac{1}{1+e^{-net}} \quad (2)$$

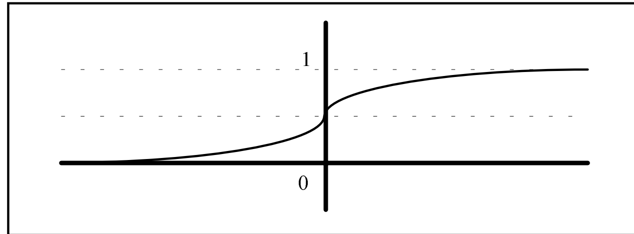

Figure 2.2 Sigmoid Function

A multi-layer feedforward neural network consists of an input layer of non-computational units (one for each input), one or more hidden layers of computational units and an output layer of computational units. Backpropagation is the standard training method that is applied to multi-layer feedforward networks. The algorithm consists of passing the input signal forward and the error signal backward through the network. In the forward pass one input vector is presented to the input layer and the input signal is filtered forward through the subsequent layers of the network to generate a set of outputs. The network outputs are compared with the actual outputs of the training example and an error signal is generated. In the backward pass, the synaptic weights are updated using the learning constant according to the effect they have in generating the incorrect outputs (if the network output matches the actual output, the error signal has no effect in adjusting the synaptic weights). Figure 2.3 demonstrates the passing of signals through a neural network and the structure of a multi-layer network. The presentation of training examples to the network is repeated until the network is stabilized and no more adjustments are required for the synaptic weights or the maximum number of epochs has been reached.

The training of a neural network by backpropagation is also viewed as a gradient descent search to find the synaptic weights that will yield the global minimum error [1]. However, the error surface of a multi-layer network may contain local minima that may cause the network training to conclude prematurely. One way to overcome this problem of error convergence is to retrain the network with the same data and a different set of initial synaptic weights.

In our software testing experiments, we use a *1-of-n* encoding of outputs, where each output unit of a neural network represents a possible value of the output variable. The output unit having the highest-valued activation function ("the winning output") is taken as the network prediction for the output of the tested program. As indicated by [8], the difference between the outputs can be used as a measure of the confidence in the network prediction.

As the number of input and output units are fixed according to the data used to train the network, the number of units per additional hidden layer and the number of hidden layers used have to be determined. Likewise, the learning constant and the number of epochs also have to be decided upon. These four

variables are used as "tuning knobs" to fine-tune the performance of the network and to minimize the final misclassification error.

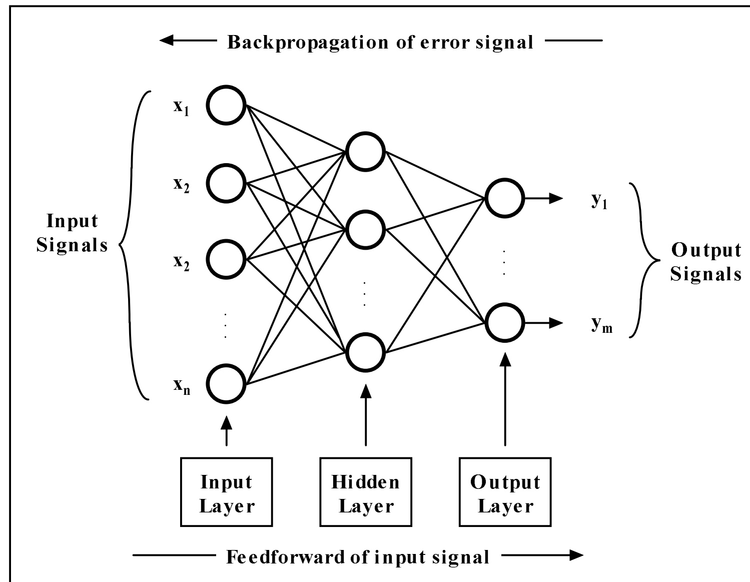

Figure 2.3 Structure of a Multi-layer Network and Signal Propagation Through the Network

Multi-layer feedforward neural networks are capable of solving general and complex problems and the backpropagation technique is computationally efficient as a training method [8]. In this paper, we use the multi-layer network trained by backpropagation to simulate a credit approval application.

3 DESCRIPTION OF THE TESTING METHODOLOGY

Our testing methodology can be viewed as a black-box approach. In the training phase (see Figure 3.1.), the input vector (test case) for each training example is generated randomly, subject to the specifications of the tested program. Each input vector is then provided as an input to the *original version* of the tested program, which subsequently generates a corresponding output vector. The input and the output vectors of the program are used for training the neural network. The trained neural network can perform the function of an automated "oracle" [12] for testing the subsequent versions of the program (a process known as “regression testing”).

In the evaluation phase (see Figure 3.2), each test case is provided as an input vector to a *new version* of the tested program and to the trained ANN. For each input vector, the comparison tool calculates the absolute distance between the winning ANN output and the corresponding value of the application output. All the outputs are numbers between zero and one. The computed distance is then placed into one of three intervals defined by two threshold values in the [0, 1] range. If the network prediction and the application output are identical and the computed distance falls into the interval between 0.0 and the low threshold (the distance is very small), this means that the network prediction matches the program output and both outputs are likely to be correct. In this case, the comparison tool reports that the program output is correct. On the other hand, if the network prediction is different from the application output and the distance between outputs is very large (falls into the interval between the high threshold and 1.0), a program fault (incorrect output) is reported by the comparison tool. In both cases, the network output is likely to be correct and it is reliable enough to evaluate the correctness of the

application output. However, if the distance is placed into the interval between the low threshold and the high threshold, the network output is considered unreliable. In the last case, the comparison tool reports a program fault only if the network prediction is identical to the application output.

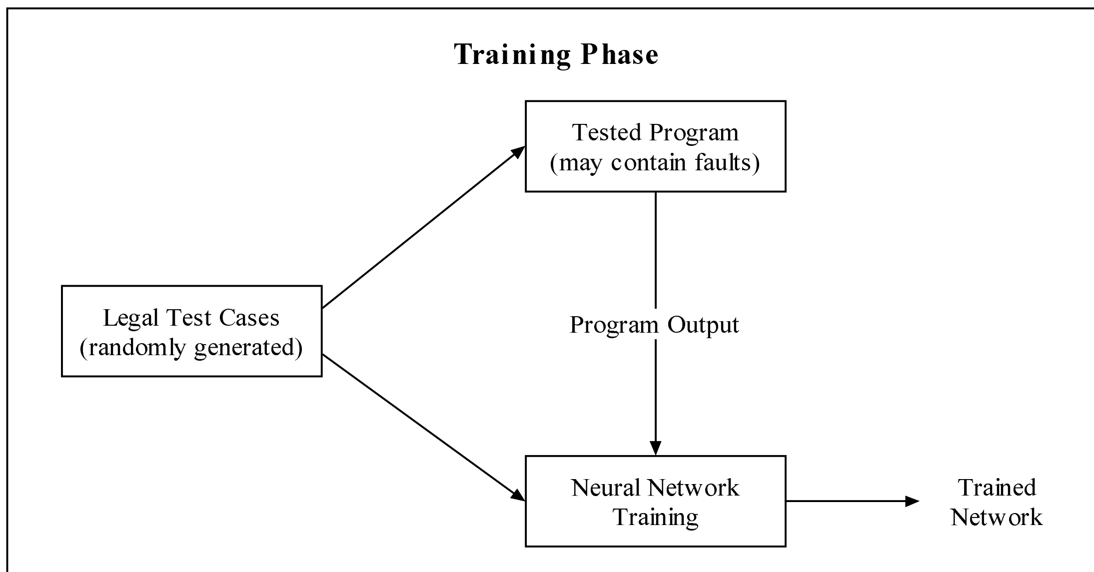

Figure 3.1: Overview of the Training Phase

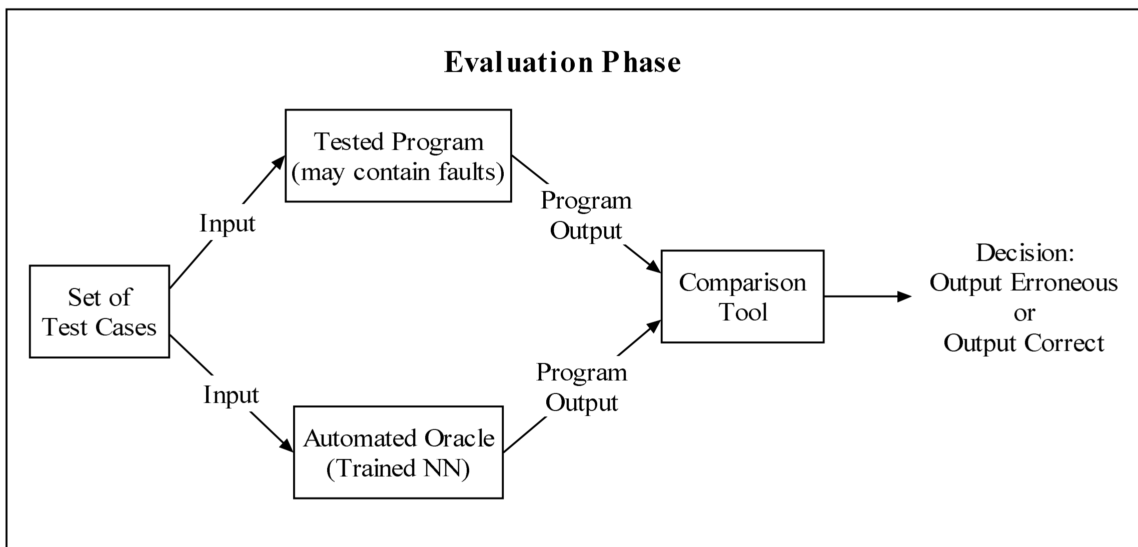

Figure 3.2: Overview of the Evaluation Phase

The comparison tool is employed as an independent method of comparing the results from the neural network and the results of the faulty 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.

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 the 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.

		Faulty Application Output	
		Correct	*Wrong*
ANN Output	*Correct*	1 True Positive	2 True Negative
	Wrong	4 False Negative	3 False Positive

Table 3.3 Each Output has a Defined Category

Table 3.3 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 to be 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 we are mainly interested in finding the wrong outputs (categories 2 and 3), we also note that there is 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 one hundred percent correctly due to the problem of error convergence discussed in Section 2. In the next section, we show that despite this seeming disadvantage the percentage of incorrectly classified outputs is minor compared to the overall data set size. Thus, we are interested in the cases where the tested application output is wrong: categories 2 and 3 using the notation of Table 3.3. When the outputs are compared with one another, they are the 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.

		Comparison of outputs	
		Same	*Different*
Type of Output	*Binary*	• Both correct • Both wrong	• ANN correct • Faulty application correct
	Continuous	• Both correct • Both wrong	• ANN correct • Faulty application correct • Both Wrong

Table 3.4 Possibilities of Fault Types

The two types of outputs are handled differently by the comparison tool, as there are four possible situations for the binary output and five for the continuous output. The analysis of the continuous output differs in one part with respect to the binary output. When both the ANN output and the faulty application are different, the ANN output may be correct, the faulty application output may be correct or in the case of a continuous output, they both may be wrong, whereas only two situations are possible for the binary output (see Table 3.4). In our experiment (see next section), the values of both the low and the high threshold for each type of output were obtained experimentally to yield the best overall fault detection results.

4 DESCRIPTION OF THE EXPERIMENT

The design of the experiment is separated into three parts, one for the sample application, one for the neural network, and the other for the comparison tool that calculates the distance between the two outputs (See Figure 1.1). The specification and the algorithm for a credit card approval application are used to provide the necessary format of the input and output attributes and the placement of the injected faults. The design of the neural network is also partially dependent on the input/output format of the application; however, the tuning knobs for training the neural network were manually determined by trial and error. Figures 3.1 and 3.2 present the flow of the process followed in the experiment. In the training phase, the input data for the training of the neural network and the tested program is generated randomly. The input data is fed singularly to the tested program as an input vector that generates an output vector for each training example. Thus, the set of input vectors and output vectors are passed as input to the two-layer neural network, which is trained using the backpropagation algorithm. In the evaluation phase a mutated version of the tested program is created by making one change at a time to the original program. The testing data is then passed to both the neural network and the tested program. The outputs from both the tested program and the trained network are then processed by the comparison tool that makes a decision on whether the output of the tested program is erroneous or correct.

The process that the experiment follows begins with the generation of test cases. The input attributes are created using the specification of the program that is being tested while the outputs are generated by executing the tested program. The data undergoes a preprocessing procedure in which all continuous input attributes are normalized (the range is determined by finding the maximum and minimum values for each attribute) and the binary inputs and output are either assigned a value of 0 or 1. The continuous output is treated in a different manner and the output of each example is placed into the correct interval specified by the range of possible values and the number of intervals used. The processed data is used as the data set for training the neural network. The network parameters are determined before the training algorithm begins. The training of the network includes presenting the entire data set for one epoch and the number of epochs for training is also specified. The backpropagation training algorithm concludes when the maximum number of epochs has been reached or the minimum error rate has been achieved. The network is then used as an "oracle" to predict the correct outputs for the subsequent regression tests. As explained in Section 3 above, the comparison tool uses both the data from the oracle and the tested program to evaluate whether the output of the latter is wrong or correct.

Design of the 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 is used throughout this paper is 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 4.1 and Table 4.2 provides a sample data set. Each record includes nine input attributes and two output attributes (one binary, one continuous).

Name of the attribute	**Data Type**	**Attribute Type**	**Details**
Serial ID Number	integer	Input	unique for each customer
Citizenship	integer	Input	0: American 1: Others
State	integer	Input	0: Florida 1: other states
Age	integer	Input	1-100
Sex	integer	Input	0: Female 1: Male
Region	integer	Input	0-6 for different regions in US
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
Number of dependents	integer	Input	0-4
Marital status	integer	Input	0: Single 1: Married
Credit approved	integer	Output	0: No 1: Yes
Amount	integer	Output	>= 0

Table 4.1 Input Attributes of the Data

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

Table 4.2 Sample Data Used During Training (before Preprocessing)

For example, customer 2 of Table 4.2 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 $1200. Since a neural network can be trained only on numeric values, all categorical attributes (citizenship, state, etc.) were converted to numeric form. The training data consists of 500 records (test cases); the additional 1000 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 was sufficient data to find faults in the tested program.

```
1. If ((Region == 5) || (Region == 6))
2.   Credit Limit = 0
3. Else
4.   If (Age < 18)
5.      Credit Limit = 0
6.   Else
7.      If (Citizenship == 0){
8.         Credit Limit = 5000  + 1000 * Income Class
9.         If (State == 0)
10.           If ((Region == 3) || (Region == 4))
11.              Credit Limit = Credit Limit * 2.00
12.           Else
13.              Credit Limit = Credit Limit * 1.50
14.        Else
15.           Credit Limit = Credit Limit * 1.10
16.        If (Marital Status == 0)
17.           If (Number of dependents > 0)
18.              Credit Limit = Credit Limit + 200 * Number of Dependents
19.           Else
20.              Credit Limit = Credit Limit + 500
21.        Else
22.           Credit Limit = Credit Limit + 1000
23.        If (Sex == 0)
24.           Credit Limit = Credit Limit + 500
25.        Else
26.           Credit Limit = Credit Limit + 1000
27.  }
28.   Else{
29.        Credit Limit = 1000 + 800 * Income Class
30.        If (Marital Status == 0)
31.           If (Number of dependents > 2)
32.              Credit Limit = Credit Limit + 100 * Number of Dependents
33.           Else
34.              Credit Limit = Credit Limit + 100
35.        Else
36.           Credit Limit = Credit Limit + 300
37.        If (Sex == 0)
38.           Credit Limit = Credit Limit + 100
39.        Else
40.           Credit Limit = Credit Limit + 200
41. }
42.     If Credit Limit = 0
43.        Credit Approved = 1
44.     Else
45.        Credit Approved = 0
```

Figure 4.3 Algorithm for the Credit Card Approval Application

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 backpropagation training algorithm. The algorithm that the application follows can be found in Figure 4.3. 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 in 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. Table 4.4 provides a listing of the details for the injected faults.

Description of the neural network

The backpropagation neural network used in this experiment is capable of generalizing the training data, however the network cannot be applied to the raw data as the attributes do not have uniform

presentation. Some input attributes are not numeric and in order for the neural network to be trained on the test data, the raw data has to be preprocessed and as the values and types differ for each attribute it becomes necessary that the input data is normalized. The values of the continuous attribute vary over a range while there are only two possible values for the binary attribute (0 or 1). Thus for the network to be able to process the data uniformly, the continuous input attributes have to be normalized to a number between 0 and 1. The data is preprocessed by normalizing the values for all the input attributes according to the maximum and minimum possible values for each attribute. The output attributes were processed in a different manner. If the output attribute was binary, the two possible network outputs are 0 and 1. On the other hand, continuous outputs cannot be treated in the same way, since they can take an unlimited number of values. To overcome this problem, we have divided the range (found by using the maximum and minimum possible values) of the continuous output into ten equal-sized intervals and placed the output of each training example into the correct interval (a value between 0 and 9).

The architecture of the neural network is also dependent on the data. In this experiment, we used eight non-computational input units for the eight relevant input attributes (the first is not used as it is a descriptor for the example) and twelve output computational units for the output attributes. The first two output units are used for the binary output. For the purposes of training, the unit with the higher output value is considered to be the "winner". Similarly the remaining ten units are used for the continuous output. The initial synaptic weights of the neural network were obtained randomly and covered a range between –0.5 and 0.5. Experimenting with the neural network and the training data, we concluded that one hidden layer with twenty-four units was sufficient for the neural network to approximate the original application to within a reasonable accuracy. A learning rate of 0.5 was used and the network required 1500 epochs to produce a 0.2 % misclassification rate on the binary output and 5.4 % for the continuous output. Figure 4.5 shows the number of epochs vs. the convergence of the error.

Fault #	Line #	Original Line	Injected Fault	Error Type	Output 1 (Credit Approval)		Output 2 (Credit Limit) Final root mean squared error
					Error (%)	Confidence Interval (95%)	
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
3	1		if ((*Region* == 4) || (*Region* == 5))	Argument Change	15.80	[14.6, 17.00]	0.93
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	[79.5, 81.90]	4.12
6	4		if(*Age* < 25)	Argument Change	9.20	[8.30, 10.10]	0.73
7	7	If (*Citizenship* == 0)	if(*Citizenship* == 1)	Argument Change	3.50	[2.90, 4.10]	1.62
8	9	If (*State* == 0)	if (*State* == 1)	Argument Change	3.50	[2.90, 4.10]	0.82
9	10	If ((*Region* == 3) || (*Region* == 4)	if (*Region* == 3)	Operator Change & Argument Change	3.50	[2.90, 4.10]	0.38
10	10		if ((*Region* == 3) && (*Region* == 4))	Operator Change	3.50	[2.90, 4.10]	0.46
11	10		if ((*Region* == 2) || (*Region* == 3))	Argument Change	3.50	[2.90, 4.10]	0.41
12	10		if ((*Region* == 1) || (*Region* == 2))	Argument Change	3.50	[2.90, 4.10]	0.69
13	16	If (*Marital Status* == 0)	if (*Marital Status* == 1)	Argument Change	3.50	[2.90, 4.10]	0.85
14	17	If (*Number of dependents* > 0)	if (*Number of dependents* == 0)	Operator Change	3.50	[2.90, 4.10]	0.57
15	17		if (*Number of dependents* < 0)	Operator Change	3.50	[2.90, 4.10]	0.53
16	23	If (*Sex* == 0)	if (*Sex* == 1)	Argument Change	3.50	[2.90, 4.10]	0.94
17	30	If (*Marital Status* == 0)	if (*Marital Status* == 1)	Argument Change	3.50	[2.90, 4.10]	0.71
18	31	If (*Number of dependents* > 2)	if (*Number of dependents* >= 2)	Operator Change	3.50	[2.90, 4.10]	0.52
19	31		if (*Number of dependents* < 2)	Operator Change	3.50	[2.90, 4.10]	0.58
20	31		if (*Number of dependents* <= 2)	Operator Change	3.50	[2.90, 4.10]	0.58
21	37	If (*Sex* == 0)	if (*Sex* == 1)	Argument Change	3.50	[2.90, 4.10]	0.52

Table 4.4 List of Faults That were Tested

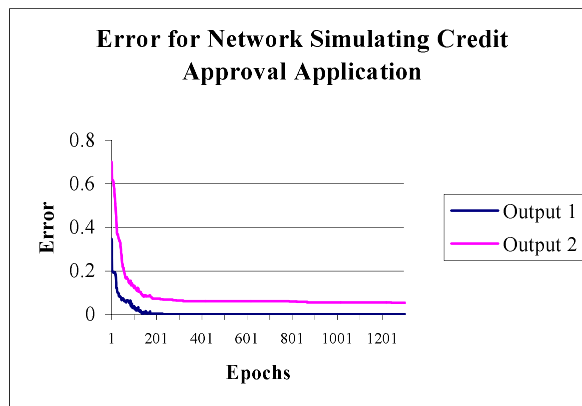

Figure 4.5 Error Convergence

5 RESULTS OF THE EXPERIMENT

The result of applying the faults are displayed in Tables 5.1, 5.2 and 5.3. The first two tables display results for the binary output (*credit approved*). Table 5.1 summarizes the results for the error rate as a function of the threshold values and Table 5.2 displays the results for the minimum average error rate. The last table summarizes the results for the minimum error rate of the continuous output (*credit amount*). The tables include the injected fault number (see Table 4.4), the number of correct outputs and incorrect outputs as determined by the "oracle", and the percentages for the correct outputs classified as being incorrect and incorrect outputs classified as being correct. The percentages were obtained by comparing the classification of the "oracle" with that of the original version of the application. The original version is assumed to be fault-free and is used as a control to evaluate the results of the comparison tool. The best thresholds were chosen to minimize the overall average of two error rates.

As can be verified from Table 4.4, the binary output was affected by an injected fault only in 5 faulty versions out of 21 due to the structure of the application: the first output is only affected by the first two conditions in the program. For this reason, only five injected faults are presented in Tables 5.1 and 5.2. In Table 5.1, the average error rates were obtained by varying the threshold values for the classification of the binary output. Table 5.2 shows the lowest error rate obtained as a function of the thresholds. Comparing the overall average error rates in Tables 5.1 to the minimum value in Table 5.2, there is not much of a difference between them. However, in comparing the values for every set of threshold values in Table 5.2, the individual error percentages vary for each fault.

	Injected Fault Number	Low Threshold = 0.21 High Threshold = 0.81		Low Threshold = 0.21 High Threshold = 0.91		Low Threshold = 0.31 High Threshold = 0.81		Low Threshold = 0.31 High Threshold = 0.91	
		Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)	Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)	Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)	Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)
	02	1.40	16.43	1.40	12.86	1.16	16.43	1.16	12.86
	03	1.41	14.97	2.11	13.61	1.29	15.65	1.99	14.29
	04	2.22	5.56	3.75	5.07	2.22	5.80	3.75	5.31
	05	6.74	2.80	12.92	2.80	5.62	3.16	11.80	3.16
	06	1.40	31.88	1.61	27.54	1.29	31.88	1.50	27.54
Percentage Average		**2.63**	**14.33**	**4.36**	**12.37**	**2.32**	**14.58**	**4.04**	**12.63**
Total Average		**8.48**		**8.37**		**8.45**		**8.34**	

Table 5.1 The Error Rate as a Function of the Threshold Values for the Binary Output

	Injected Fault Number	Number of Correct Outputs	Number of Incorrect Outputs	Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)
	02	860	140	1.28	14.29
	03	853	147	1.88	13.61
	04	586	414	3.41	5.07
	05	178	822	10.11	2.80
	06	931	69	1.61	28.99
Percentage Average				**3.66**	**12.95**
Total Average				**8.31**	

Table 5.2 The Minimum Error Rate for the Binary Output (Low Threshold = 0.26, High Threshold = 0.86)

Table 5.3 summarizes the results for the continuous output. Due to the increased complexity involved in evaluating the continuous output, there is a significant change in the capability of the neural network to distinguish between the correct and the faulty test cases: the minimum average error of 8.31 achieved for the binary output vs. the minimum average error of 20.79 for the continuous output. An attempt to vary the threshold values also did not result in an evident change to the overall average percentage of error for the continuous output.

	Injected Fault Number	Number of Correct Outputs	Number of Incorrect Outputs	Percentage of Correct Outputs Classified as Being Incorrect (%)	Percentage of Incorrect Outputs Classified as Being Correct (%)
	02	140	860	28.14	1.43
	03	307	693	6.78	49.51
	04	587	413	5.33	21.12
	05	822	178	8.99	3.89
	06	69	931	23.63	13.04
	07	559	441	11.11	5.72
	08	355	645	21.86	7.89
	09	217	783	8.17	73.27
	10	303	697	7.60	52.15
	11	238	762	7.74	66.39
	12	276	724	24.17	10.51
	13	371	629	23.05	6.47
	14	99	901	22.86	23.23
	15	65	935	23.32	33.85
	16	407	593	23.27	4.91
	17	273	727	22.56	13.55
	18	20	980	24.49	50.00
	19	71	929	24.54	1.41
	20	1000	0	0.00	4.20
	21	125	875	20.91	50.40
Percentage Average				**16.93**	**24.65**
Total Average				**20.79**	

Table 5.3 The Minimum Error Rate for the Continuous Output (Low Threshold = 0.10, High Threshold = 0.90)

6 CONCLUSIONS

In this experiment, we have used a neural network as an automated "Oracle" 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 has a good coverage of the input range. The backpropagation 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 the 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.

ACKNOWLEDGEMENTS

This work was partially supported by the USF Center for Software Testing under grant no. 2108-004-00.

REFERENCES

1. C. Anderson, A. von Mayrhauser, R. Mraz, On the Use of Neural Networks to Guide Software Testing Activities, Proceedings of ITC'95, the International Test Conference, October 21-26, 1995.
2. J. Choi, B. Choi, Test Agent System Design, 1999 IEEE International Fuzzy Systems Conference Proceedings, August 22-25, 1999.
3. R. A. DeMillo, A. J. Offutt, Constraint-based Automatic Test Data Generation, IEEE Transactions on Software Engineering SE-17, 9 (Sept. 1991), pp. 900-910.
4. S. Haykin, Neural Networks, A Comprehensive Foundation, 1999.
5. T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, S. J. Aud, Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System, IEEE Transactions on Neural Networks, vol 8, num 4, July 1997, pp. 902-909.
6. T. M. Khoshgoftaar, R.M. Szabo, Using Neural Networks to Predict Software Faults During Testing, IEEE Transactions on Reliability, vol 45, num 3, September 1996, pp. 456-462.
7. L. V. Kirkland, R. G. Wright, Using Neural Networks to Solve Testing Problems, IEEE Aerospace and Electronics Systems Magazine, vol 12, num 8, August 1997, pp. 36-40.
8. T. Mitchell, Machine Learning, McGraw-Hill, 1997.
9. S. A. Sherer, Software Fault Prediction, Journal of Systems and Software, vol 29, num 2, May 1995, pp. 97-105.
10. J. M. Voas, G. McGraw, Software Fault Injection, 1998.
11. J. M. Voas, K.W. Miller, Software Testability: The New Verification, IEEE Software, 1995.
12. E. Weyuker, T. Goradia, A. Singh, Automatically Generating Test Data from a Boolean Specification, IEEE Transactions on Software Engineering SE-20, 5(May 1994), pp. 353-363.