Use neural network to improve fault injection testing

Yichen Wang
Science & Technology on Reliability & Environment Engineering Laboratory, Beihang University
Beijing, China
wangyichen@buaa.edu.cn

Yikun Wang
Antares Testing International Ltd.
Beijing, China
ehco@antares-testing.com

***Abstract*—fault injection is an effective technique in software testing. By introducing faults to software under test, fault injection can improve the coverage of a test, as the same time, the fault injected in software contributes significantly to find true fault related to fault injected. In this paper we propose a software testing method based on fault injection. In this method, we first use neural network to calculate the value of fitness function that describe the proximity of two paths, and then use simulated annealing algorithm to generate test data. In this method we use the value of fitness function as the criteria in simulated annealing algorithm. Experiment shows that this method can improve the efficiency of generating test data obviously.**

Keywords— software fault injection; neural network; software testing; fault propagation path; simulated annealing algorithm

I. INTRODUCTION

Software is changing, the size of the software is getting bigger, the quality features of the software are very complicated and contain many functions, and the structure of the software grows significantly complex. It's so difficult to predict the behavior of the certain software, even for the one that is designed and programmed by ourselves[1]. In such cases, the complexity of the software poses a huge challenge for software verification and validation, especially, software testing for highly-reliable software is facing unprecedented difficulties.

Fault injection test is an effective way to verify the fault tolerance of highly reliable software. Fault injection has been used in mutation testing primarily as a mechanism for evaluating the adequacy of test data [2]. The use of software fault injection can improve the coverage of software test, and shorten the test time and reduce test costs. When we use a designed set of faults to mutate the program under test, the design criteria for test cases becomes to cover these known faults as much as possible and to discover new faults [3]. According to the basis of test case design, software testing technology is usually divided into black box test and white box test. At present, although both black box test and white box test are used in fault injection test, but for white box test, we are doing the test on code directly, so it's much easier to inject the fault, so as to find the errors. [4].

White-box testing concerns the degree to which the test cases exercise or cover the logic (source code) of the program under test. Among all the adequacy criteria for white-box testing, path coverage is the most important and most studied one. In the path-based test case generation process, a typical path PT in Control Flow Graph (CFG) of program is obtained as the target path, and then input data of program $X \in D$ (input dimension of program) is obtained by genetic algorithm or other artificial intelligence method, so that PE is the path that program executed when X is input. If PE and PT coincide, X will be the generated test case[5][6]. In fault injection test, we specify the fault propagation path as the coverage criteria of test case design. The fault propagation path is an executable path that contains a known injection fault in CFG of the program [7]. Design the test case with the fault propagation path as the coverage criterion can not only improve the efficiency of the fault injection test, but also have a very good effect on the discovery the correlation fault of complex software [8]. However, for complex software, the number of paths of the software is huge, and the relationship between the input data and the execution path is very complex and uncertain. In order to improve the efficiency and accuracy of test case, using the intelligent algorithm to generate the complex software test data becomes the hot spot of the research in recent years[9]. In this paper, we use the annealing algorithm to design test cases that can cover the fault propagation path, and introduce the neural network to predict the fitness value of the path, which greatly improves the efficiency of the annealing algorithm.

we introduce the related work in section II, and introduce some basic concept in section III. In section IV, we design a neural network to predict the value of fitness function, and then we propose a method that generate test data based on annealing algorithm using neural network. Finally, we complete the experiments to verify the effectiveness of this method.

II. RELATTED WORK

A. Software Fault Injection Testing

Fault injection technology is a non-traditional software testing technology. According to a specific fault model, it produces faults in an artificial and conscious way, and imposes specific faults in the software under test to speed up the process of exploring software error. Fault injection technology was first applied to hardware testing in the 1970s, and later was developed based on hardware technology, software-based technology, simulation-based technology and ion radiation technology based on different test objects. Software Fault Injection Test (SFIT) refers to the use of software to test the software system technology. It first began in 1978 DeMillo proposed program mutation test. SFIT can improve the quality of software and evaluate the software level. It is widely used in

978-1-5386-2072-4/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS-C.2017.69

some areas such as high-reliability software and safety-intensive software, such as aviation, aerospace software test, etc. SFIT is very effective for fault tolerance test of high-reliability software. According to the stage of the fault injected, SFIT is divided into static injection and dynamic injection. The static injection is widely used in white box test.

The first step of SFIT is to select the set of fault injected. The quality of SFIT depends to a certain extent, the quality of the set of fault injected. To simulate a variety of complex errors, the errors in complex software system can be found in testing. For example, simulating errors in specifications will be able to find the software design-related errors; for embedded systems, real-time systems, concurrent systems and distributed systems which contain complex time logic, simulating time logic error will be able to effectively detect time-related errors. When we use SFIT to test software, the principle of test case design is to cover the injected fault as more as possible. The reason for this is based on the following: 1) when the injected fault is triggered, the fault tolerance of the software can be tested and evaluated; 2) when the injected fault is triggered, it may activate the correlation fault associated with the injected fault, and the correlation fault is a kind of fault that is very difficult to be found in the complex software; 3) when the injected fault is triggered, the software will be executed in some paths that is difficult to achieve, and then improve the coverage of the test[11][12][13][14][15].

B. Neural Networks

Neural network is a mathematical model or computational model that mimics the structure and function of biological neural networks for estimating or approximating functions. At present, the application of neural network in software testing is mainly embodied in three aspects: First, the neural network as a classifier to predict the ability to retest test data, according to the selection of thin test data; Second, the neural network as a system approximate, instead of the real test software to implement the test. Third, use the neural network for the test data generated.

Aggarwal et al. studied how to solve the Oracle problem using neural network. First, according to the specification of the program under test, the input and output pairs of the program under test were obtained and used as training samples of the neural network. Then, using the training samples, neural network model can be used as the application model; for a given input data, through the neural network to predict its expected output. Anderson et al. studied how to use neural networks for error prediction. The main work includes training the neural network, and using the trained network to predict errors. In the training phase, they use samples to train the neural networks. In their neural network model, input is test data, while the output is the type of error. In the predicting phase, they use the neural network to predict the type of the error that might be discovered. Fu Bo presents a technique of dynamic self-organizing neural networks to automatically generate test data for revealing software faults. The technique consists of two parts: the first one is niche genetic algorithm, which generates a small initial fault test data set in software input domain; the second one is dynamic self-organizing feature map, which can repeatedly generate lots of test data for finding faults by using initial fault test data set[16][17][18][19].

C. Simulated Annealing Algorithm

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. It is a random-search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system. In 1953, Metropolis first proposed the idea of simulated annealing applied to intelligent algorithms. By 1983, this idea was successfully used to solve the problem of combinatorial optimization problems. After extensive application, due to the outstanding performance of the simulated annealing algorithm in solving the local minimum problem, many researchers have begun to study it deeply, which makes the algorithm greatly developed.

At present, as a kind of intelligent algorithm, Simulated Annealing Algorithm(SAA) is gradually applied to design test cases in software testing. Mohammed presented a method that combined genetic algorithm and simulated annealing algorithm. Firstly, the genetic algorithm is used to evolve the population solution, and then the simulated annealing algorithm adjusts the optimal solution. Li Xiaoqing presented a kind of software test case automated generation method based on annealing immune genetic algorithm. Test case generation model and basic idea of AIGA are introduced. The algorithm combines SAA with immune algorithm to overcome the disadvantages of both algorithms. Zhong Xiaomin proposed multiple paths test case generator based on annealing genetic algorithm. Zhong designed the fitness function and introduced storage paths mechanism to synthesize multiple test data to cover multiple target paths. Zhong also improved crossover of genetic algorithm, and applied simulated annealing to the mutation, in order to improve the efficiency of the algorithm.

For the existing simulated annealing algorithm in the application of software test cases, the processing of the solution is simply accepted by the probability, which led to the process of accepting the solution only related to formal considerations, or is very blind to find the global optimal solution outside the local optimal solution. In addition, when using the annealing algorithm to generate test data, we need to input each test data to run the program under test in order to assess the performance of the test data, so that it takes too much time on testing, and resulting in testing efficiency is very low[20][21][22][23].

III. BASIC CONCEPTS

A. Fault Injection and Fault Propagation Path

When using the fault injection technique for white-box testing, two basic rules must be understood. First, the prerequisite of finding the injected fault is that the path where contains the injected fault is executed. Second, more than one propagation path contains the same injected fault.

Before the test case design, we must first complete the construction of the fault propagation model, which requires two steps, one is to select the injected fault set, and to complete the faults of the injection; the second is to build the control flow

graph of the program under test where the faults have been injected, and get to the fault propagation path set.

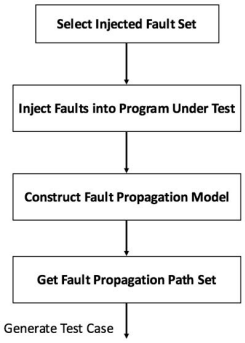

Fig. 1. Preparation for Generate Test Case

1) Software Fault Injection

- Select the typical faults. Based on the characteristics and historical fault of the software under test, these typical faults have the ability to find new faults, in particular the ability to activate correlation faults.
- Select faults that meet the fault tolerance requirements. According to the requirements in the software specifications, we should select those faults which can activate fault-tolerant module in the software under test, and such faults are usually used to verify and evaluate the ability of fault tolerance of software under test[7].

2) Fault Propagation Model

- There are many forms of software fault propagation models, such as control flow graphs, data flow graphs, and mathematical models or probability models. This paper chooses to build a fault propagation model based on a control flow graph[27][28][29].
- In the control flow graph of software with injected faults, we can get the paths that contain the statement where the fault is injected. We call it the fault propagation path. It is important to note that for each injection, there are more than one propagation path.

B. Fitness Function

In a structural test, when using test technology to generate test data that covers a particular branch or condition, the program under test is first instrumented and some randomly generated test data is executed. Subsequently, the quality of these test data is decided by the fitness function to assess the role of the search mechanism, the test data that is closer to the test target is constantly generated, and ultimately they will achieve the desired purpose. In this process, the design of the fitness function is the key to the application of searching technology to solve the problem of software testing. The fitness function is made up of two components – the approach level and the branch distance. The approach level measures how close an input vector was to execute the current structure of interest, based on how near the execution path was to reaching it in the program's control flow graph. The branch distance reflects how 'close' the alternative branch from the last critical branching node (a branching node with an exit that, if taken, causes the target to be missed) was to take. The combination between the two is the most commonly used method.

The layer proximity is the number of branch statements (excluding the bifurcation point itself) that a branch deviates from the destination path to the midpoint of the target path. The branch distance metric is initially used by Korel, which generates test data by changing the method of variables. If an unspecified path is executed when the test data is executed, then the deviation from the specified path is generated. This deviation is called the branch distance, and the search target is to minimize this deviation.

Approach_level(X) is layer proximity, which is a measure that represents the distance between a path P(*X*) and a target path P. It is the number of branch statements that a branch deviates from the destination path to the midpoint of the target path.

Branch distance is another function that can represent the distance between the specific path and the target path. It is used to describe the degree of how the input data is close to each branch. For example, there is condition in one path "if a > = 10", and then when we let the condition value be true. Our goal is to let the true branch of the statement be executed. When the program executes the statement with *X* as the input data, the value of a is *a(X)*, then the branch distance function of *X* corresponding to the true branch of the statement is

$$\text{branch\_dist}(X) = \begin{cases} 0, & a(x) \geqslant 10 \\ 10 - a(x), & \text{otherwise} \end{cases} \quad (1)$$

TABLE I. BRANCH DISTANCES FOR DIFFERENT CONDITIONS

Branch predicate	***Branch Function f***
A = B	abs(A - B)
A < B	A - B
A > B	B – A
A ≤ B	A - B
A ≥ B	B - A
A ≠ B	abs(A - B)

In general, the fitness function represented as fitness(*X*), that is sum of the layer proximity and normalized branch distance, that is:

$$\text{fitness}(X) = \text{approach\_level}(X) + \text{normal}(\text{branch\_dist}(X)) \quad (2)$$

and *normal (branch_dist)* is:

$$\text{normal}(\text{branch\_dist}) = 1 - 1.001^{-\text{branch\_dist}} \quad (3)$$

The necessary and sufficient condition for *fitness (X)* = 0 is that *X* is the same as the test target path. The smaller the value of *fitness (X)* is, the closer the *X* is to the test target, so the test data covering the test target path generates a problem that can

be converted to a function *fitness(X)* minimization problem.[24][25][26][27][18]

IV. CONTRUCTING NEURAL NETWORK

A. Structure of the Neural Network

We use a three-layered neural network to calculate the value of fitness function. The net has n inputs which is equal to the number of input data of software under test and one output which is the value of the fitness function.

It is important to determine the number of hidden nodes in the neural network design. A two-layered BP network with infinite hidden layer nodes can realize any non-linear mapping from input to output. However, for a limited number of input-to-output mappings, there is no need for an infinite hidden layer, which involves the question of how to choose the number of hidden nodes. Hidden layer nodes are often based on the experience of previous design or the experiments testers did themselves. It is generally believed that the number of hidden nodes has a direct relationship with the number of input and output units. Moreover, if the number of hidden nodes is too small, we cannot generate the number of connection weights to satisfy the learning need of the samples. Also, if the number of hidden layers is too large, the generalization ability of the network will be deteriorated.

We use the following equation to determine the number of hidden nodes[18]:

$$n_1 = \sqrt{n+m} + a \tag{4}$$

n_1 is the number of hidden nodes, n is the number of input nodes, m is the number of output nodes, and a is a random integer number which is between 1 and 10.

So, the structure of neuron is shown in fig 2.

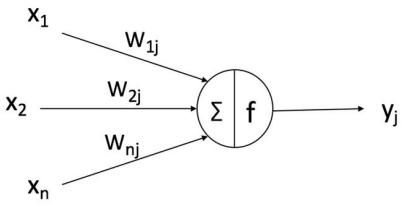

Fig. 2. Structure of single neuron

We use the equation to calculate output of j_{th} neuron in hidden layer.

$$y_i = f\left(\sum_{i=1}^{n} w_{ij} x_i\right) \tag{5}$$

In this paper we use the structure of neural network as below (fig 3):

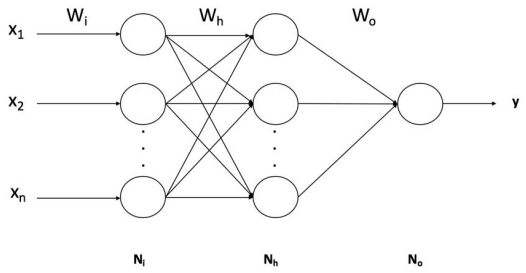

Fig. 3. Structure of neural network

In fig.3:

The input of neural network is the input of software under test, and we use X that is a matrix to present the input:

$$X = (x_1, x_2, \ldots, x_n) \tag{6}$$

We use W_i to present the weight of each input:

$$W_i = (w_{i1}, w_{i2}, \ldots, w_{in}) \tag{7}$$

The weight between the input layer and the hidden layer is represented by the matrix $W_h = [w_{ij}]$, where w_{ij} is the weight of the i_{th} neuron of the input layer to the j_{th} neuron of the hidden layer.

$$w_h = (w_{11}, \ldots, w_{ij}, \ldots, w_{nn_1}) \tag{8}$$

The weight is represented by the vector $W_o = [w_j]$, where w_j is the weight of the j_{th} neuron of the hidden layer to the output layer:

$$w_o = (w_1, w_2, \ldots, w_{n_1}) \tag{9}$$

B. Training Data Set

Generalization training is the standard way of training feed-forward networks. During training, each input X, at time t is associated with the corresponding output O_t. Thus the network learns to model the actual functionality between the independent (or input) variable and the dependent (or output) variable.

In order to train the neural network that we are building, we need a sample set of data. A sample is a binary set which includes an input and an oracle *(X, d)*, and $X= (x_1, x_2, \ldots, x_n)$ is the input vector which includes n inputs of software under test, and d is the output oracle which is the value of fitness function.

In the case of ensuring the quality and distribution of the sample data, the size of the sample data determines the accuracy of your neural network training results. The more sample data, the higher precision.

Software fault injection test is the primary technique of testing fault tolerance and finding correlation faults, usually after a requirement-based software test. In requirement-based test stage, we design test cases based on software requirement, and these test cases will help us to generate the sample set to train the neural network. The steps to get the training data set are as follows：

- Before running test cases based on software requirement, insert instruction in the program under test;
- Specify a target path in the program under test;
- Run test cases based on software requirement;
- Collect every path according to result of running test cases;
- Calculate the value of fitness function for every path;
- Combine one input data in test case and the corresponding value of fitness function to generate a sample.

After that, we will get a set of sample whose size equals to the amount of test cases based on requirement.

C. Trainning the Network

We use back propagation(BP) algorithm to train the neural network. It is a common method of training neural networks and is used in conjunction with an optimization method such as gradient descent. In BP algorithm, the error of the output is used to estimate the error of the direct preamble of the output layer, and the error of the previous layer is estimated by this error, so that the error estimation of all the other layers is obtained. BP algorithm includes two phases[30][31]:

1) Phase 1:

Each propagation involves the following steps:

- Forward propagation of a training pattern's input through the neural network in order to generate the network's output value(s).

The output of input layer is:

$$w_i x_i = (w_1 x_1, \ldots\ldots, w_n x_n) \tag{10}$$

The input of hidden layer is:

$$net_j = \sum_{i=1}^{n} w_{ij} (w_i x_i) \tag{11}$$

The output of hidden layer is:

$$o_j = f(net_j) = f[\sum_{i=1}^{n} w_{ij} (w_i x_i)] \tag{12}$$

The input of output layer is:

$$s = \sum_{j=1}^{n1} w_o o_j \tag{13}$$

The output of output layer, which is the output of the neural network, is:

$$y = f(s) = f(\sum_{j=1}^{n1} w_o o_j) \tag{14}$$

- Backward propagation of the propagation's output activations through the neural network using the training pattern target to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

In this paper, we select an error function measuring the difference between two outputs.

$$E = \frac{1}{2}(y - d)^2 \tag{15}$$

In the equation above:

- E is the square error,
- t is the oracle output for the training sample,
- y is the actual output of the output neuron
- The factor of $(1/2)$ conveniently cancels the exponent when the error function is subsequently differentiated, we use *(X, d)* to present the test data, and *X* is input, and *d* is the expected output which is called oracle. So:

$$E = \frac{1}{2}(y - d)^2 = \frac{1}{2}[f(\sum_{j=1}^{n1} w_j o_j) - d]^2$$
$$= \frac{1}{2}\{f[\sum_{j=1}^{n1} w_j f(\sum_{i=1}^{n} w_{hij} (w_i x_i))] - d\}^2 \tag{16}$$

In this paper we use a logical function as activation function, which is a nonlinear differentiable function.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{17}$$

And the partial derivative with respect to the outputs:

$$\frac{\partial f}{\partial x} = f(1 - f) \tag{18}$$

2) Phase 2:

Following below steps for each weight:

- Multiply the weight's output delta and input activation to find the gradient of the weight.
- Subtract a ratio (percentage) of the weight's gradient from the weight.

This ratio (percentage) influences the speed and quality of learning; it is called the learning rate. The greater the ratio, the faster the neuron is trained, but the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates whether the error varies directly with, or inversely to, the weight. Therefore, the weight must be updated in the opposite direction, "descending" the gradient.

To update the weight using gradient descent, we must choose a learning rate. The change in weight, which adds to the old weight, is equal to the product of the learning rate and the gradient, multiplied by -1:

In this paper, we use below equation to update the weight of output layer:

$$\triangle w_o = -\alpha_1 \frac{\partial E}{\partial w_o} \tag{19}$$

We update the weight of hidden layer as below:

$$\triangle w_{ij} = -\alpha_2 \frac{\partial E}{\partial w_{ij}} \tag{20}$$

We update the weight of input layer as below:

$$\triangle w_i = -\alpha_3 \frac{\partial E}{\partial w_i} \tag{21}$$

Repeat phases 1 and 2 until the performance of the network is satisfied.

TABLE II. THE STEPS OF TRAINING NEURAL NETWORK

```
initialize network weights (often small random values)
do
  forEach training example named ex
    prediction = f(network, ex)  // forward pass
    actual = oracle(ex)
    compute error (prediction - actual) at the output units
    compute {ΔWo} //ΔWo for all weights from hidden layer to output layer
    compute {ΔWh} //ΔWh for all weights from input layer to hidden layer
    compute {ΔWi} //ΔWi for all weights of input layer
    update network weights
until all examples finished
return the network
```

V. GENERATE TEST DATA USING SIMULATED ANNEALING ALGORITHM

In section III of this paper, we introduce how to inject faults into the software and how to get the fault propagation path of the software. In this part, we focus on the simulated annealing algorithm that can generate the test data. Simulated annealing algorithm has two main parts: the first part is that using neural network to predict the value of fitness function in order to filter the best from the initial test data that are generated randomly. The first part is an important part to improve the efficiency of the algorithm; the second part is to evaluate the efficiency of the generated test data to see if they are the optimal solution.

In simulated annealing algorithm, we use the neural network to predict the value of fitness function and to filter the value according to the threshold. This is an important criterion to judge whether the test data satisfies the fault propagation path coverage. The specific steps are as follows:

1) For each fault propagation path, we randomly generate a set of input data;
2) Use the neural network to calculate the test input corresponding to the input function value;
3) Check whether the value is within a reasonable range of threshold, if in this range, then run the program and calculate the actual adaptation function value. If it is not within a reasonable range of threshold, return to step1;
4) Determine whether the adaptation value is 0, if zero, then this group of input data can cover this fault propagation path; if the value is not 0, then enter other judgment procedure.

The basic process of annealing algorithm is shown in Fig. 4: □

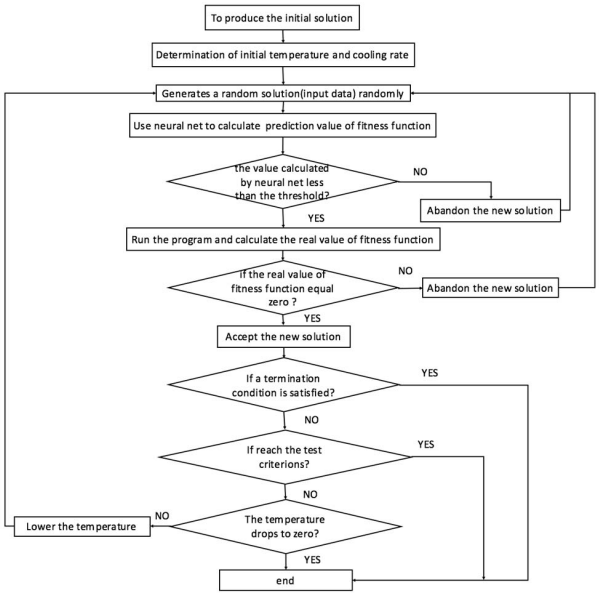

Fig. 4. Annealing algorithm for generating test data

VI. EXPERIMENT

The experiment in this paper includes two parts:

1) We design an experiment to verify the accuracy of value of fitness function calculated by neural network;

2) We design an experiment to evaluate the time cost of generating test data using neural network.

B. The program example

The problem is to judge the type of a triangle according to the relationship of the three sides. The program contains a clear and typical logic, which is widely used in the software testing literature. The inputs of the triangle problem are three reals, which are the three sides of a triangle. The output of the program is the judgment of triangle. There are several outputs: equilateral, isosceles, ordinary, non-triangular and input error.

C. Verify the effectiveness of using neural networks

Yao had done some valuable experiments about verifying the accuracy of neural network[18]. Based on her experiments, we repeat the experiment with a new program in this paper.

We chose the triangle classification program to run the experiment. First, we chose two target paths according to control flow of the program under test. Second, for each target path, randomly selected a number of test data; and then, respectively, using Plug-In (PI) procedures and Neural Networks (NN) to get the two Values of the Fitness Function (VFF). Finally, comparing the two values we obtained, we evaluate the accuracy of the neural network.

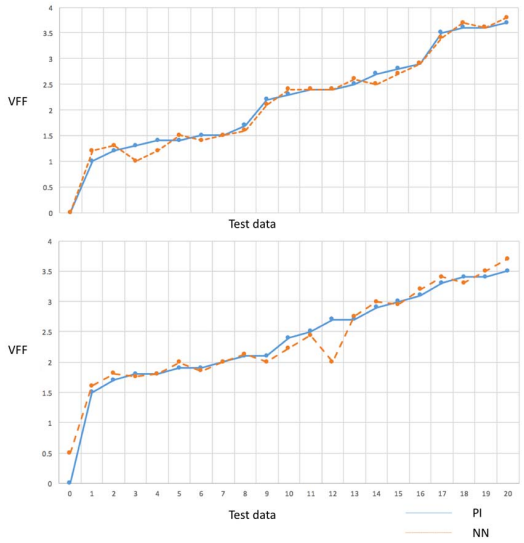

Fig. 5. effectiveness of using neural networks

We can see that the individual fitness value and the insertion method obtained by the neural network method are slightly different, but can basically reflect the individual fitness value, the overall trend of the two changes in the same way. Therefore, it is feasible to use the neural network method to simulate the individual fitness value.

D. Evaluate the time cost of different algorithm

In this experiment, we will compare the three test data generation methods, they are AA (Annealing Algorithm without using neural network), and AANN (Annealing Algorithm with using Neural Network), and GR (Generate test data Randomly).

We continue to use the triangle classification program to run the experiment. First, we inject 10 types of different faults into the program, and then get 10 fault propagation paths according to the control flow. Second, we use AA, AANN and GR to generate test data in order to cover these 10 fault propagation paths. Finally, we use the iteration number in the algorithm to compare the efficiency of three different algorithm. The lower number, the more efficient.

The results of experiment are showed in table III.

TABLE III. TEST RESULTS

	AA	**AANN**	**GR**
1	100	70	102
2	104	98	111
3	70	60	68
4	43	40	58
5	74	70	79
6	130	116	125
7	38	35	40
8	80	76	93
9	75	69	77
10	72	66	80

The data in the table above can also be expressed as the following figure

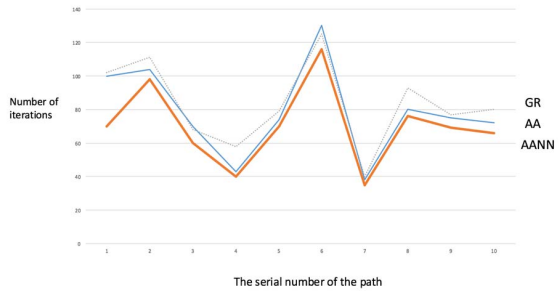

Fig. 6. Cost time of different algorithms

Results above indicate that AANN has the least time cost of these three algorithms. This shows that using neural network to the annealing algorithm can improve the efficiency of path coverage, and then improve the efficiency of fault injection testing.

VII. CONCLUSION

In this paper, we define the fitness function, and then construct the neural network to predict the value of the fitness function which is generated by the specific input data. By using the filtering of the fitness value as the judgment condition, we optimize the annealing algorithm, thus improve the fault coverage efficiency of fault propagation path in fault injection testing.

REFERENCES

[1] Peter Feiler and Kevin Sullivan, “Ultra-Large-Scale Systems: The Software Challenge of the Future” June 2006

[2] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Of- fut, and K. N. King. An extended overview of the mothra sofware testing environment. Proc. ACM SIGSOFT/IEEE □ Second Workshop on Software Testing, Verification, and Analysis, pages 142–151, July 1988

[3] CHEN Jin-Fu+, LU Yan-Sheng, XIE Xiao-Dong,” Research on Software Fault Injection Testing ” Journal of Software,2009,20(6):

[4] Beizer B. Software Testing Techniques. 2nd., New York: John Wiley & Sons, Inc., 1990. □

[5] Mayer,art of software testing

[6] Hitesh Tahbildar, AUTOMATED SOFTWARE TEST DATA GENERATION: DIRECTION OF RESEARCH

[7] K. Wang and Y. C. Wang, "A Software Testing Method Based on Error Propagation Slicing Technology," *2015 International Conference on Computer Science and Applications (CSA)*, Wuhan, 2015, pp. 241-245

[8] Xun Wang and Yichen Wang, “Research Progress on Error Propagation Model in Software System,” Computer Science, 2016, 43(6):1-9, 27

[9] Kevilienuo Kire and Neha Malhotra, “Software Testing using Intelligent Technique”, International Journal of Computer Applications Volume 90– No.19, March 2014

[10] Jean A, Martine A, Louis A, Yves C, Jean-Charles F, Jean-Claude L, Eliane M, David P. Fault injection for dependability validation: A methodology and some applications. IEEE Trans. on Software Engineering, 1990,16(2):166• 182.

[11] Carreira J, Madeira H, Silva JG. Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Trans. on Software Engineering, 1998,24(2):125• 136.

[12] Ghani AK, Nasser AK, Jacob AA. FERRARI: A flexible software-based fault and error injection system. IEEE Trans. on Computers, 1995,44(2):248• 260.

[13] Goswami KK. DEPEND: A simulation-based environment for system level dependability analysis. IEEE Trans. on Computers, •1997,46(1):60−74. •

[14] Looker N, Munro M, Xu J. Simulating errors in Web services. Int'l Journal of Simulation Systems, Science, 2004,5(5):29−37. •

[15] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. IEEE Computer, •1978,11(4):34−41. •

[16] Aggarwal KK, Singh Y, Kaur A, Sangwan OP. A neural net based approach to test oracle. ACM Software Engineering Notes, 2004, 29(3):1−6. [doi: 10.1145/986710.986725] •

[17] Anderson C, Mayrhauser AV, Mraz RT. On the use of neural networks to guide software testing activities. In: Proc. of the IEEE Int'l Test Conf. on IEEE Computer Society Test Technology Technical Committee and IEEE Philadelphia Section. Washington, 1995. 720−729. [doi: 10.1109/TEST.1995.529902]

[18] Yao XJ, Gong DW, Li B. "Evolutional test data generation for path coverage by integrating neural network," Journal of Software, 2016,27(4):828-838(in Chinese)

[19] Fu Bo. "Automatic Software Functional Test Data Generation Based on Dynamic Self-organizing Neural Networks". Acta Aeronautica et Astronautica Sinica. 2006,27(5):888-892

[20] R.A. Rutenbar. "Simulated annealing algorithms: an overview". IEEE Circuits and Devices Magazine. 1989, 5(1) : 19 - 26

[21] Zhong Xiaomin, Wu Xiaolin, Zhao Xuefeng. "Multiple paths test cases generator based on annealing genetic algorithm". Application on Research of Computer, 2010,27(12):4544-4547(in Chinese)

[22] Mohammad Al-Fayoumi, P.K. Mahanti, Soumya Banerjee. "OptiTest:optimizing test case using hybrid intelligence", Proc of World Congress on Engineering, 2007:43-48.

[23] Li Xiaoqing, Zhang Wenxiang. "Test Case Generation Based on Annealing Immune Genetic Algorithm". Computer Simulation. 2008, 25(5):171-174（in Chinese）

[24] Neelam Gupta, Aditya P. Mathur, and Mary Lou Sofia. Automated Test Data Generation Using An Iterative Relaxation Method. In *proceeding of the 6th ACM SIGSOFT international symposium on Foundations of software engineering; 231-244/98, FSE 1998*

[25] *Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A Unified Fitness Function Calculation Rule for Flag Conditions to Improve Evolutionary Testing. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 337-341 ASE. 2005*

[26] *Jin-Cherng Lin, Pu-Lin Yeh. Automatic test data generation for path testing using Gas. Information Sciences 131; 47-64, 2001*

[27] *Paulo Marcos Siqueira Bueno and Mario Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In proceedings of 15th International conference on Automated Software Engineering, 209-218, ASE 2000* •

[28] *Andrey Morozov, Klaus Janschek, "Probabilistic error propagation model for mechatronic systems ", Mechatronics 24 (2014) 1189–1202*

[29] *Xiaocheng Ge, Richard F. Paig, "Probabilistic Failure Propagation and Transformation Analysis ", International Conference on Computer Safety, Reliability, and Security SAFECOMP 2009, 215-228*

[30] *N. Karunanithi, D. Whitley and Y. K. Malaiya, "Using neural networks in reliability prediction,"* IEEE Software*, vol. 9, no. 4, pp. 53-59, July 1992.*

[31] http://neuralnetworksanddeeplearning.com/chap2.html

[32] Kun Wang,Yichen Wang,Liyan Zhang. Software testing method based on improved simulated annealing algorithm[M]，2014: 418-421