Reliability Engineering and System Safety 170 (2018) 73–82

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

Software reliability prediction using a deep learning model based on the RNN encoder–decoder

Jinyong Wang[a,*], Ce Zhang[b]

[a] School of Software Engineering, Shanxi University, Taiyuan, China
[b] School of Computer Science and Technology, Harbin Institute of Technology at Weihai, Weihai, China

A R T I C L E I N F O

Keywords:
Deep learning model based on RNN encoder–decoder
Model comparison
Neural network models
Parameter models
Software reliability

A B S T R A C T

Different software reliability models, such as parameter and non-parameter models, have been developed in the past four decades to assess software reliability in the software testing process. Although these models can effectively assess software reliability in certain testing scenarios, no single model can accurately predict the fault number in software in all testing conditions. In particular, modern software is developed with more sizes and functions, and assessing software reliability is a remarkably difficult task. The recently developed deep learning model, called deep neural network (NN) model, has suitable prediction performance. This deep learning model not only deepens the layer levels but can also adapt to capture the training characteristics. A comprehensive, in-depth study and feature excavation ultimately shows the model can have suitable prediction performance. This study utilizes a deep learning model based on the recurrent NN (RNN) encoder–decoder to predict the number of faults in software and assess software reliability. Experimental results show that the proposed model has better prediction performance compared with other parameter and NN models.

1. Introduction

With the rapid development of software technology, modern software sizes have become larger and more software functions must be attained to satisfy client requirements. Assessing software reliability is an important issue in the modern software development process. Researchers have developed different software reliability growth models to solve this problem. These models can be generally divided into two categories, namely, parameter models [1–9] and non-parameter models [10–13]. However, no single model can accurately predict the fault number in software in all testing conditions.

Researchers have developed many parameter models in the past 40 years to assess software reliability, such as those based on non-homogeneous Poisson process, stochastic differential equation, and Bayes process. Parameter models must generally estimate external parameters by utilizing least squares or maximum likelihood estimation. Thus, different estimation methods can yield different external parameter estimation results. These methods also require assumptions established in advance to build the relevant parameter models. Several of these assumptions are consistent with the realistic testing process, whereas others are not in the realistic testing scenarios. Thus, the assumptions adopted in a testing case must be different from those in other testing cases because of the complexity of software testing. Furthermore, proposing the assumptions to include all testing circumstances is almost impossible. The assumptions proposed in parameter models are subjective, limited, and have incomplete testing feature selection to simplify and facilitate modeling. Proposing objective and realistic assumptions in parameter models remains a challenging task.

Other software reliability models have also been employed to assess software reliability, such as Bayesian Network approaches, statistical time series prediction approaches, and function approximation. However, these methods have their disadvantages as well in terms of software reliability assessment. For example, Bayesian network approaches require more fault data, complex analysis, and calculation, and some fault data must utilize subjective probability. Statistical time series prediction approaches focus on the time factor and do not consider the influence of other external factors. Therefore, the scenario of detected faults in the test exhibits a large deviation when it changes over time. The statistical time series prediction method has better performance in short-term than long-term prediction. It also shows suitable fitting performance for interpolation in function approximation but may not have suitable prediction performance in extrapolation.

Given the disadvantages of the parameter and other software reliability models, researchers have considered non-parameter models to build software reliability models, such as the neural network (NN) model [14] and a triple exponential smoothing method [13]. Karunanithi et al. [11] proposed connectionist models to predict software reliability and

* Corresponding author.
E-mail addresses: wangjinyong818@163.com (J. Wang), zhangce@hitwh.edu.cn (C. Zhang).

https://doi.org/10.1016/j.ress.2017.10.019
Received 23 March 2017; Received in revised form 18 October 2017; Accepted 21 October 2017
Available online 25 October 2017
0951-8320/© 2017 Elsevier Ltd. All rights reserved.

Acronyms

NN	neural network
RNN	recurrent neural network
DBN	deep belief network
ASTs	abstract syntax trees
DNN-RED	deep neural network model based on the RNN encoder–decoder or the proposed model
AE	average error
AB	average bias
FFN	Feed-forward network

applied four NN models to predict the fault number in software. A comparison with other parameter models indicates that the NN models have better prediction performance. However, NN models have a few disadvantages that are slightly difficult to solve. These disadvantages are as follows. 1) They can easily over-fit, their parameters are more difficult to tune, and they require numerous training tricks. 2) Training is slower, and the prediction effect is the same as that in other methods in the case of fewer levels (less than or equal to 3). Thus, utilizing NN models to predict software reliability is unrealistic because the prediction performance of these models suffers from the aforementioned problems. The accuracy of NN models in predicting software reliability still needs to be improved.

NNs have been extensively studied in recent years, and their prediction performance has been largely improved. Hinton and Salakhutdinov [15] specifically proposed a deep NN model called a restricted Boltzmann machine to identify images with a two-layer network. Hinton et al. [16] proposed a fast learning algorithm to accelerate the slow characteristics of NN in training. Srivastava et al. [17] also proposed a simple method that addresses the overfitting problem for NNs. Thus, the deep NN model has both the NN characteristics and its own advantages. The deep NN model has input, hidden (multi-layer), and output layers, which only connect between the units in adjacent layers. The same layer and cross-layer units are not connected to one another, similar to that with NNs, but they employ different training mechanisms. NNs utilize the back propagation method to train the entire NN. Their disadvantages include gradient diffusion and converging to a local minimum problem. The deep belief network adopts a layer-wise method to train the entire networks. The deep belief network is initially trained layer-to-layer from bottom to top, and then the next layer is trained utilizing a greedy algorithm. They are then fine-tuned from top to bottom utilizing a back propagation method called a wake–sleep algorithm.

Why can the deep NN model adapt better in assessing software reliability than traditional NN models? First, traditional NN models randomly set the initial value to obtain weights. Hence, the feature selection of software faults is unstable and unfixed. The deep NN model can capture the stable and accurate features of software faults through a greedy algorithm. Second, obtaining a local optimal solution in traditional NN models is generally easy. However, the deep NN model can obtain a global optimal solution through a greedy algorithm. It can also automatically learn better feature representations from software faults than other methods. Thus, the deep NN model has better prediction performance than traditional NN models.

We utilize a deep NN model in this study to predict the fault number in software and assess software reliability. Experimental results indicate that the deep NN model has better prediction performance than traditional NN and parameter models. The experimental results are also encouraging and significant for practical application and scientific research.

The contributions of this study are as follows:

(1) To the best of our knowledge, we are the first to utilize the deep NN model to predict the fault number in software and assess software reliability.
(2) Experimental results indicate that the prediction performance of the deep NN model is better than those of traditional NN and parameter models.
(3) Experimental results also prove that the deep NN model adapts better in predicting the fault number in software and assessing software reliability. It also has better robustness than traditional NN models.

The remainder of this paper is structured as follows. In Section 2, related works are introduced and discussed. Section 3 introduces the formal descriptions, background, developed process, and practical utilized method of the deep NN model based on the recurrent NN (RNN) encoder–decoder. Section 4 describes the experimental process and discusses the model comparison results. Section 5 discusses the threats to the validity of the proposed deep NN model. Conclusions are drawn in the last section.

2. Related works

Recently, recurrent neural network was popularly applied the issue of sequence to sequence. For example, Graves [18] proposed a method to predict a discrete data point and next-step real-valued sequences using long short-term memory recurrent neural network. Sutskever et al. [19] used deep neural networks to improve a general sequence learning problem and proposed an end-to-end (mapping sequence to sequence) learn method for machine translation. Pascanu et al. [20] studied a few deep recurrent neural networks and proposed a new framework for a deep recurrent neural network. Cho et al. [21] proposed the recurrent neural network (RNN) Encoder-Decoder model to improve machine translation, and used one RNN to encoder or map a sequence to a fixed vector, and used the other RNN to decoder symbols to a sequence. For software defect prediction, Wang et al. [22] used deep belief network (DBN) model to capture semantic characteristics from program abstract syntax trees (ASTs) and used this model to predict or identify software faults. Hu et al. [23] merged fault detection and fault correction processes, and used recurrent neural networks to model them. They also used genetic algorithm to optimize network configuration for improving fault prediction.

In addition, other non-parameter approaches also were used to predict software faults. For example, Wang et al. [24] used deep belief network to automatically learn semantic features from source code for defect prediction, and the results show that the proposed model can better capture fault features compared with other models. Tian and Noore [25] proposed an evolutionary neural network model to predict software cumulative failure time, and used genetic algorithm to search a global solution for the neural network architecture. Bai et al. [26] used Markov Bayesian network to dynamically predict software failure occurrence. Costa et al. [27] proposed a new approach based on genetic programming and boosting techniques to improve software reliability prediction. Li et al. [28] proposed two combination models based on Adaboosting approach for software reliability prediction. Torrado et al. [29] proposed a Bayesian model based on Gaussian processes to predict the number of software faults. Wei et al. [30] considered the different trend between the test data and the training data, and proposed a new support vector regression approach based on particle filter algorithm. Roy et al. [31] proposed a neuro-genetic approach considering logistic growth curve in the artificial neural network, and used genetic algorithm to optimize the network weights.

From above literature discussions, recurrent neural network have been successfully applied in sequence to sequence problems. In this paper, we consider the fault detection time as an input time series, and regard the cumulative number of detected faults as an output sequence. Furthermore, we use RNN Encoder-Decoder model to capture the features from fault datasets, and predict next-step and end-point fault number. The experimental results indicate that RNN Encoder-Decoder model can be used to accurately predict the number of faults in software and

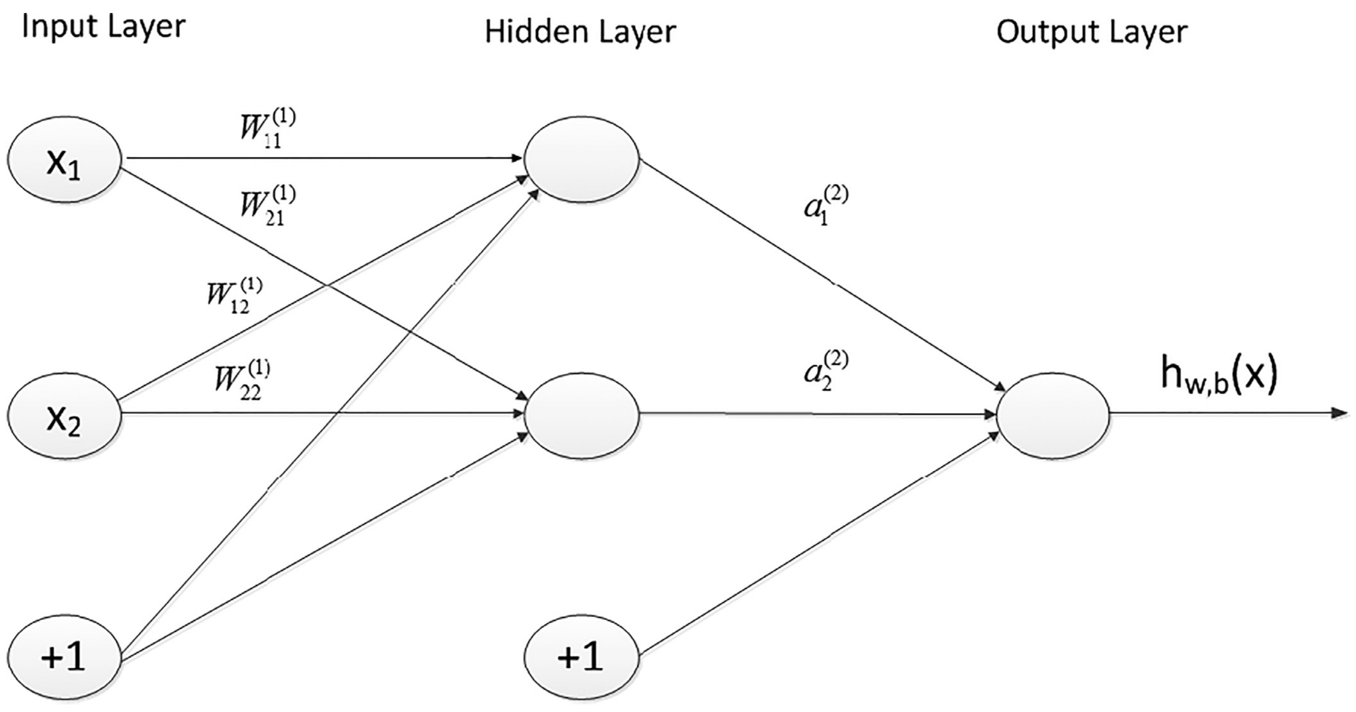

Fig. 1. Plots of a three-layer NN, including the input, hidden, and output layers.

evaluate software reliability. In addition, when the detected faults are more and more, software reliability will be higher and higher. In other words, when more faults are detected, the probability of software failures is getting lower. Therefore, Accurately predicting the number of software faults is very important for ensuring the reliability of software.

3. Deep NN model

3.1. Formal descriptions

The deep NN model can be represented as a system *S:* $I \geq S_1 \geq S_2 \geq ..... \geq S_k \geq O$ with *n* layers. S_i represents the *i*-th layer, *I* denotes an input, and *O* denotes an output. If we adjust the system parameters and let input *I* be equal to or approximately equal to output *O*, then we can automatically obtain access to a series of input characteristics, namely, S_1, S_2,..., S_k. Thus, given an input sequence i_1, i_2,..., i_t and a corresponding output sequence o_1, o_2,..., o_t up to the current time *t*, an input i_{t+d} in a future time $t+d$ through a series of characteristic changes (S_1, S_2,...,S_k) can provide the output o_{t+d}. $d=1$ denotes the next-step prediction, while $d=n$ represents the end-point prediction in our case. The prediction problem can be formulated as follows:

$$P : \left(\left(I_t, O_t\right), i_{t+d}\right) \geq S_1 \geq S_2 \geq ... \geq S_k \geq o_{t+d}$$

where (I_t, O_t) presents a training sample or model up to time *t*, i_{t+d} represents an input in a future time $t+d$, and o_{t+d} denotes a prediction output in time $t+d$. Once we complete a training model for a deep NN, we can utilize this deep NN model to predict the fault number in software and assess software reliability.

3.2. Background: NN model

Given that a deep NN is transformed from an NN, we first introduce the NN concept. The NN is generally composed of input, hidden, and output layers. The hidden layer can have multiple layers. Every layer can have a plurality of neurons or units. For example, Fig. 1 shows a three-layer NN. The input layer includes two neurons or units, the hidden layer includes two neurons or units, and the output layer includes one neuron or unit.

Suppose that $W^{(l)}$ and $b^{(l)}$ denote the weights and biases at layer l, $a_i^{(l)}$denotes the activation (or output vector) of unit *i* at layer l, and $h_{w,b}(x)$ denotes the output value. The NN in Fig. 1 can be represented as follows:

$$a_1^{(1)} = x_1 \quad a_2^{(1)} = x_2 \tag{1}$$

$$a_1^{(2)} = f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + b_1^{(1)}) \tag{2}$$

$$a_2^{(2)} = f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + b_2^{(1)}) \tag{3}$$

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + b_1^{(2)}) \tag{4}$$

$$f(x) = \frac{1}{1+\exp(-x)} \tag{5}$$

where *f(x)* represents a sigmoid function or logistic regression function, which can also be called an activation function.

Eqs. (1)–(5) can be rewritten as follows:

$$\begin{cases} Z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)} \\ a_i^{(2)} = f(Z_i^{(2)}) \\ h_{W,b}(x) = a_i^{(3)} = f(Z_i^{(3)}) \\ f(Z) = \dfrac{1}{1+\exp(-Z)} \end{cases} \tag{6}$$

We can utilize $a_i^{(l+1)}$as the prediction value at layer $l+1$ in Eq. (6).

We can utilize a square error function to compute the weights $W^{(l)}$ and biases $b^{(l)}$. For example, given the sample ($x^{(1)}$, $y^{(1)}$, $x^{(2)}$, $y^{(2)}$,...$x^{(n)}$, $y^{(n)}$) and NN output values $h_{W,b}(x)$, the object function can be denoted as follows:

$$J(W, b; x, y) = \frac{1}{2}\left\|h_{w,b}(x) - y\right\|^2 \tag{7}$$

$$J(W, b) = \frac{1}{n}\sum_{i=1}^{n} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\|W\|_F^2,$$
$$\|W\|_F^2 = \sum_{l=1}^{L}\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l+1}} W_{ij}^{(l)} \tag{8}$$

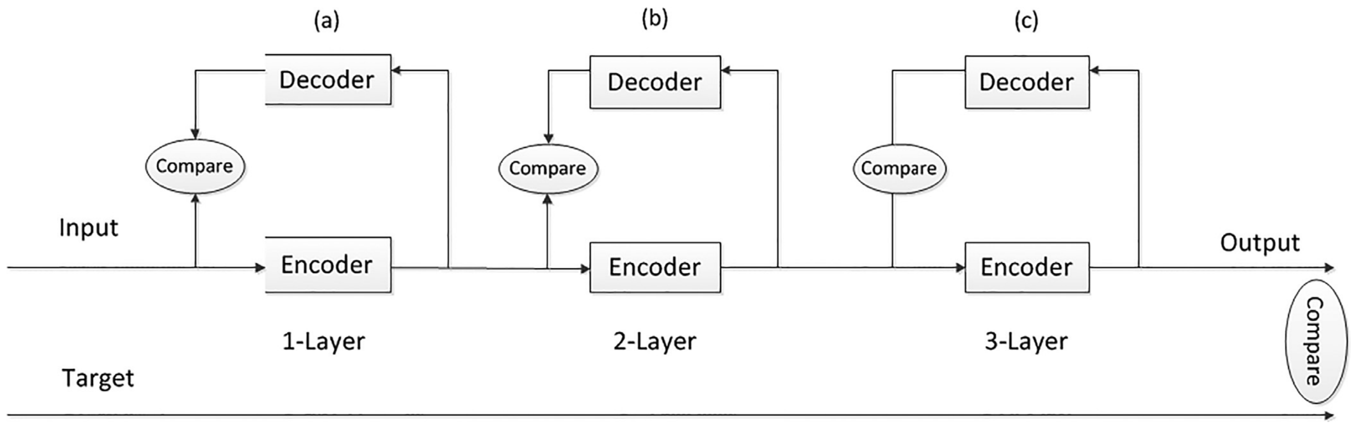

Fig. 2. Plots of the feature learn process for the deep-learning NN.

where L is the layer number, $\frac{\lambda}{2}\|W\|_F^2$ denotes a regularization term that can prevent overfitting, F indicates F-norm of matrix W, and λ is a weight decay parameter. $J(W, b; x, y)$ denotes a single training sample (x, y) in Eq. (7). $J(W, b)$ denotes an overall training sample $(x^{(1)}, y^{(1)}, x^{(2)}, y^{(2)},\ldots x^{(n)}, y^{(n)})$.

We can compute the partial derivatives of Eq. (8) as follows:

$$\begin{cases} W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \dfrac{\partial J(W,b)}{\partial W_{ij}^{(l)}} \\ b_i^{(l)} := b_i^{(l)} - \alpha \dfrac{\partial J(W,b)}{\partial b_i^{(l)}} \end{cases} \tag{9}$$

where α represents a learn rate. We can compute the parameters W and b of Eq. (8) through the iteration of Eq. (9).

3.3. Model developed by a deep NN

Aside from the NN and deep NN having the same portions, the deep NN also has a unique method that automatically captures the training features. Fig. 2 shows that the deep NN is capturing the relevant features of software faults in the training process. The said figure assumes a three-layer NN. Fig. 2(a) shows the first layer training process. We enter an input to an encoder and obtain a code. The code is then placed into a decoder and outputs a message. If no information is lost after the conversion, then the output message is theoretically equal to the input. Therefore, we can adjust the encoder and decoder parameters by comparing the output message with the input and obtaining the first input signal representation (i.e., the first code). We can then obtain the first feature. We can repeat this process through layer-by-layer training and obtain all the features of the three layers. Fig. 2(a)–(c) show these processes. Finally, we can obtain all trained features and utilize these features to predict the fault number in the software by comparing the target samples and fine-tuning the entire system. This process is called end-to-end learning. Fig. 2 show the entire fine-tuning process. The said figure shows that feature extraction is obtained automatically.

The input is approximately equal to the message by constructing the encoder and decoder. This process can be regarded as an unlabeled learning process. However, the final step in Fig. 2 can be regarded as a labeled learning process.

3.4. RNN encoder–decoder

This study employs the deep learning model based on the RNN encoder–decoder to predict the fault number in software and assess software reliability in software testing. Thus, this section introduces RNN encoder–decoder theory.

The RNN encoder–decoder was proposed by Cho et al. [21] and Sutskever et al. [19] to encode an input sentence into a vector and then decode the vector into another sentence. The RNN encoder–decoder includes two aspects: the encode process [i.e., the input sequence $(X_1, X_2,\ldots, X_T)$ through the encoder is converted to vector V], and the decode process [i.e., vector V that passes through the decoder is converted to the output sequence $(Y_1, Y_2,\ldots, Y_T)$]. The basic structure of the RNN encoder–decoder is shown in Fig. 3 [23].

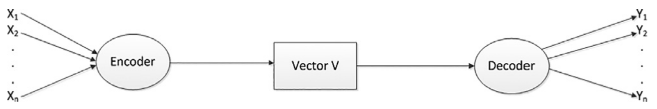

Fig. 3. Plots of the encoder and decoder process for the recurrent NN.

The input sequence $(X_1, X_2,\ldots, X_T)$ in RNN is inputted to the encoder. The hidden state of the current time is determined by the state of the previous and current time inputs. This sequence can be presented as follows:

$$H_t = F\left(H_{t-1}, X_t\right) \tag{10}$$

Generating the final vector V by gathering every hidden layer information yields the following:

$$V = Q\left(H_1, H_2, \ldots, H_T\right) \tag{11}$$

where H_t is a hidden layer state at time t in RNN, both F and Q are nonlinear functions, and V is a vector that includes different hidden states in an RNN.

The decoding process is an inverse encoding process in the RNN. It can be applied to predict the next output Y_{t+1} through vector V and after the predicted output $\{Y_1, Y_2,\ldots, Y_t\}$. This process can be denoted as follows:

$$Y_{t+1} = \arg\max_{\theta} P(Y_{t+1}) = \prod_{t=1}^{T} P(Y_{t+1}|\{Y_1, Y_2, \cdots, Y_t\}, V) \tag{12}$$

It can also be written in the RNN as follows:

$$P(Y_{t+1}|\{Y_1, Y_2, \ldots, Y_t\}, V) = G(Y_t, H_{t+1}, V) \tag{13}$$

where P denotes a probability, G represents a nonlinear function or can be regarded as multi-layered NNs, and θ denotes several model parameters. The input sequence through the model trained by the RNN encoder–decoder can generate the predictive output sequence.

3.5. Deep learning model utilized

This section introduces how to adopt the deep NN model based on the RNN encoder–decoder, which we refer to as the proposed model and abbreviated as DNN–RED, to predict the fault number in the software.

Our adopted deep NN model based on the RNN encoder–decoder has four layers, namely, one input layer, two hidden layers, and one

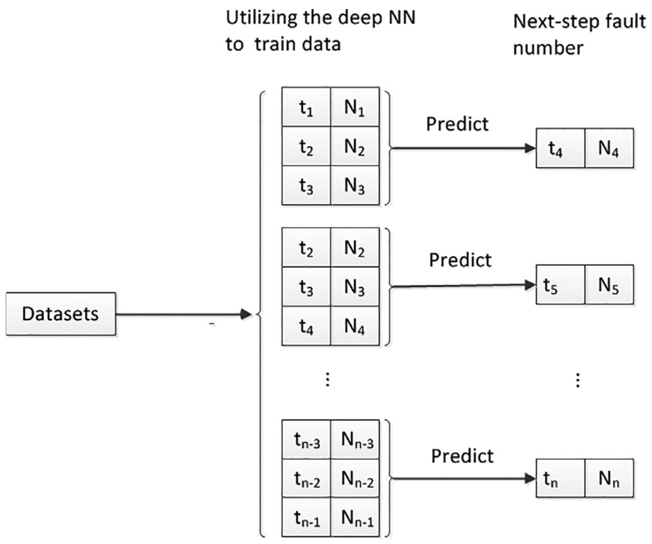

Fig. 4. Plots of the next-step fault prediction process for the proposed deep NN.

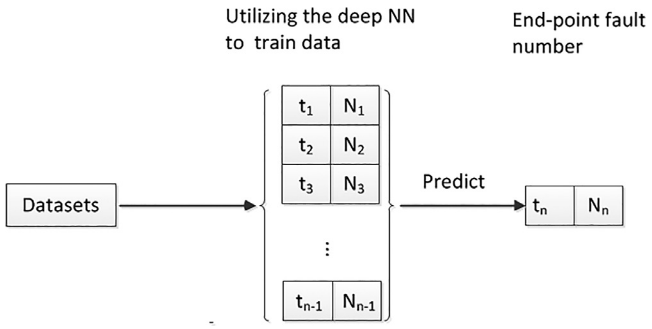

Fig. 5. Plots of the end-point fault prediction process for the proposed deep NN.

output layer. Given that the range of software failure data values can be extremely large and the data values are unevenly distributed, we then scale software failure data values into the range of 0.1–0.9 to accurately predict the fault number in the software. Considering that the software fault prediction belongs to time series prediction, we take a detected time of faults as input and take an accumulative number of detected faults as output in the training. We also employ the pair (t_i, o_i) and i∈1, 2,…, n to train the deep-learning NN and t_{i+1} to predict o_{i+1}, which is called the next-step prediction. In addition, we utilize t_n to predict o_n, which is called the end-point prediction. Note that we apply the accumulative fault number as output values. When the predicted accumulative number of next software failure occurrence is less than the current accumulative number of fault occurrence, then this prediction is discarded because it is a practically impossible case. We take the average of multiple output results as the final prediction.

In Figs. 4 and 5, t_i denotes the time of detected faults, and N_i presents the accumulative number of detected faults by time t_i. Fig. 4 shows that a few pairs (t_i, N_i) are trained by the deep NN to predict the next-step fault number N_{i+1} at time t_{i+1}. Fig. 5 shows that the pair (t_i, N_i), where i is from 1 to n-1, is trained by the deep NN to predict the end-point fault number N_n at time t_n.

In this paper, we use three pairs (I_{t-2},O_{t-2}), (I_{t-1},O_{t-1}) and (I_t,O_t) to train the model, and use the trained model to predict next-step value, that is o_{t+1}. In addition, we use pairs (I_1,O_1), (I_2,O_2),…, (I_{t+n-1},O_{t+n-1}) to train the model, and use the trained model to predict end-point value, that is o_{t+n}.

4. Numerical examples

We perform several experiments in this section to compare the deep NN model with four NN models and five parameter models. We introduce the utilized historical fault data sets, several compared models, and model comparison criteria before the experiments. We utilize 14 fault data sets and 3 model evaluation criteria from literature [11] to effectively compare the performance of all the models.

4.1. Historical fault data sets

A total of 14 different fault data sets from literature [11] were employed to compare the performance of all the models. The first fault data set (DS1) was recorded in [32]. A total of 27 faults were detected from a compiler project with approximately 1000 code lines at the execution time of 794 (CPU time). The second fault data set (DS2) was collected from a real-time command and control system in [1]. Approximately 136 faults were detected at the execution time of 88,682 (CPU time), and the system had approximately 21,700 instructions. The third fault data set (DS3) was reported in [33]. A total of 46 faults were detected from an on-line data entry control software package with approximately 40,000 code lines for 21 days.

The fourth fault data set (DS4) was collected from a PL/I database application software with approximately 1317,000 code lines in [33]. A total of 328 faults were detected at the execution time of 47.65 (CPU time). The fifth fault data set (DS5) was collected from three software modules of a hardware control program with approximately 35,000 code lines in [33]. The accumulative number of detected faults in software testing for 13 months was 279. The sixth fault set (DS6) was from [34]. A total of 3207 faults were detected from an application program with approximately 2400,000 code lines for 13 months. The seventh fault data set (DS7) was collected from a real-time control application with approximately 870,000 code lines in [35]. A total of 535 faults were detected in software testing for 109 days. The eighth fault data set (DS8) was reported in [36]. A total of 481 faults were detected from a monitoring and real-time control system with approximately 200,000 code lines for 111 days.

The ninth fault date set (DS9) was collected from [36]. A total of 55 faults were detected from a railway interlocking system with approximately 14,500 instructions for 199 days. The 10th fault data set (DS10) was reported in [36]. The accumulative number of detected faults from a real-time control system with approximately 90,000 code lines was 198 for 16 days. The 11th fault data set (DS11) was collected from a flight dynamic application with approximately 10,000 code lines in [37]. A total of 118 faults were detected for 82.36 h. The 12th fault data set (DS12) was reported in [37]. A total of 180 faults were detected from a flight dynamic application with approximately 22,500 code lines for 162.32 h. The 13th fault data set (DS3) was recorded in [37]. A total of 213 faults were detected from a flight dynamic application with approximately 38,500 code lines for 216.07 h. The 14th fault data set (DS14) was collected from a control application system in [35]. A total of 266 faults were detected in testing for 46 days.

4.2. Compared models

We select different software reliability models, including parameter and non-parameter models, to effectively compare our deep NN model based on the RNN encoder–decoder. The parameter models include a logarithmic model, an exponential model, a power model, a delayed S-shape model, and an inverse polynomial model. Except for the inverse polynomial, the other parameter models are non-homogeneous Poisson process models. Moreover, they are all two-parameter models. The non-parameter models include four NN models from [11], such as Feed-forward network (FFN)-Generalization, FFN-Prediction, JordanNet-Generalization, and JordanNet-Prediction. All comparison models utilized in the present study are the same as those in [11]. More detailed information is presented in [11].

4.3. Model evaluation criteria

We require several criteria to quantify the model comparison results in prediction and effectively assess the model prediction performance. This study applies two model comparison criteria, namely, average error (AE) and average bias (AB) proposed in [11,38], as well as one measure to rank the models.

Criterion 1.

$$AE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{U_{ij} - D_j}{D_j}\right| \tag{14}$$

where U_{ij} denotes the accumulative number of estimated faults by time t_j, D_j denotes the accumulative number of observed faults by time t_j, and n is the sample size. The lower the AE values are, the better the model prediction performance will be.

Criterion 2.

$$AB = \frac{1}{n}\sum_{i=1}^{n}\frac{U_{ij} - D_j}{D_j} \tag{15}$$

where U, D, and n are the same symbols denoted in Eq. (10). $j = i + 1$ is called the next-step prediction, while j = n is called the end-point prediction in Eqs. (10) and (11).

AE can also measure the model prediction performance for the entire test phase, whereas AB can measure the model bias. When AB obtains a positive value, then the predictive value of the model is overestimated. When AB obtains a negative value, then the predictive value of the model is underestimated. If the AE value is equal to the AB value, then the model is consistent; otherwise, it is inconsistent. When a model shows a consistent bias, it has better stability than a model with an inconsistent bias [39].

Criterion 3.

$$R_m = \frac{1}{N}\sum_{s=1}^{N} AE_s^m \tag{16}$$

where R_m represents the mean of AE for the model m, N denotes the number of fault data sets, and s is an index of fault data sets. The smaller the value of R_m, the better its prediction performance becomes.

4.4. Prediction performance analysis

This section compares the proposed model with NN and other parameter models. We utilize the model comparison results in [9] to compare our results with those from previous studies. We employ the 14 fault data sets to compare the prediction performance of all the models in the present study. All the models utilized in this study are ranked in Tables 1–3 according to their respective AE values. All the models are ranked in Table 4 according to their respective R_m values. Note that Tables 1–3 show the overall average prediction error on the end-point prediction and next-step prediction for the entire testing phase. The results shown in Tables 1–3 are after training.

Table 1
Model comparison results for DS1–DS5.

Data set	Models	End-point predictions			Next-step predictions		
		AE	RANK	AB	AE	RANK	AB
1	FFN-generalization	6.63	5	−5.64	9.37	9	+0.59
	FFN-prediction	3.76	4	−1.69	5.61	3	+2.96
	JordanNet-generalization	3.05	3	−1.82	6.79	5	+2.63
	JordanNet-prediction	2.68	2	−2.20	4.66	2	−0.49
	Logarithmic	16.84	6	−16.84	7.33	6	−3.26
	Inverse Polynomial	19.40	8	−14.39	9.21	8	−0.80
	Exponential	28.35	9	−28.35	6.67	4	−4.64
	Power	18.35	7	+5.48	11.15	10	+2.27
	Delayed S-shape	35.78	10	−35.78	8.03	7	−4.25
	DNN-RED	0.07	1	+0.07	0.45	1	+0.45
2	FFN-generalization	3.52	5	+3.52	3.95	10	+3.89
	FFN-prediction	2.32	2	+2.32	2.50	9	+1.53
	JordanNet-generalization	3.11	3	−1.72	1.49	2	+0.09
	JordanNet-prediction	3.21	4	−0.3.20	2.05	8	−1.92
	Logarithmic	3.83	7	−1.75	1.33	3	−0.31
	Inverse polynomial	3.98	6	+2.02	1.39	4	+0.13
	Exponential	8.57	8	−8.32	1.33	5	−0.95
	Power	3.94	9	+2.07	1.45	6	+0.14
	Delayed S-shape	10.99	10	−10.99	1.58	7	−1.42
	DNN-RED	0.12	1	+0.12	0.42	1	+0.42
3	FFN-generalization	10.24	4	−10.24	8.44	9	v8.44
	FFN-prediction	12.32	5	−12.32	6.84	6	−5.64
	JordanNet-generalization	6.96	2	−6.96	6.84	6	−3.63
	JordanNet-prediction	9.52	3	−9.52	6.03	3	−1.54
	Logarithmic	12.48	6	−8.82	7.78	7	−7.75
	Inverse polynomial	13.29	8	−11.91	6.17	4	−1.75
	Exponential	15.87	9	−12.57	7.85	8	−7.22
	Power	12.95	7	+1.35	6.55	5	−0.52
	Delayed S-shape	27.10	10	−27.10	5.99	2	−2.09
	DNN-RED	0.11	1	+0.11	0.59	1	+0.59
4	FFN-generalization	2.45	3	−0.08	5.28	3	+1.15
	FFN-prediction	1.91	2	+0.35	4.64	2	−0.14
	JordanNet-generalization	3.79	4	−2.14	8.84	9	+2.11
	JordanNet-prediction	3.86	5	−3.10	6.11	6	−1.87
	Logarithmic	16.15	6	−9.53	5.93	4	−1.75
	Inverse polynomial	24.92	9	+8.49	7.95	8	−0.35
	Exponential	20.99	8	−14.28	6.01	5	−2.65
	Power	34.14	10	+21.61	9.4	10	+1.50
	Delayed S-shape	20.87	7	−15.37	6.25	7	−3.71
	DNN-RED	0.14	1	+0.14	0.28	1	+0.28
5	FFN-generalization	5.22	4	+5.15	10.00	8	+6.31
	FFN-prediction	6.64	5	+6.47	6.95	5	+6.44
	JordanNet-generalization	3.38	2	+0.87	5.09	2	+0.44
	JordanNet-prediction	4.99	3	+1.79	8.67	6	+0.69
	Logarithmic	7.21	6	−0.69	6.42	4	−2.29
	Inverse polynomial	32.38	9	+28.13	9.71	7	+3.96
	Exponential	10.73	7	−9.89	6.15	3	−4.25
	Power	36.35	10	+36.33	23.36	10	+17.16
	Delayed S-shape	25.89	8	−3.35	10.90	9	−0.23
	DNN-RED	0.01	1	−0.01	0.04	1	+0.04

(1) Table 1 shows that the DNN-RED has the best prediction performance among all the models for DS1, DS2, DS3, DS4, and DS5. The AE values of the DNN-RED are the lowest for DS1, DS2, DS3, DS4, and DS5 for end-point predictions and next-step predictions. Except for DS5, the AE values of the DNN-RED are consistent with the AB values for DS1, DS2, DS3, and DS4 in terms of end-point predictions. The DNN-RED model ranks first among all the models for DS1, DS2, DS3, DS4, and DS5.

(2) From Table 2, we can see that the DNN-RED has better prediction performance than the other models for DS6, DS7, DS8, DS9, and DS10. The DNN-RED has the lowest AE values for DS6, DS7, DS8, DS9, and DS10 for end-point predictions and next-step predictions. Except for DS6, the AE values of the DNN-RED are consistent with the AB values for DS6, DS7, DS8, DS9, and DS10 in terms of next-step predictions. Table 2 shows that the DNN-RED ranks first among all the models for DS6, DS7, DS8, DS9, and DS10.

(3) As can be seen from Table 3, the DNN-RED has better prediction performance than the other models for DS11, DS12, DS13, and DS14. The DNN-RED has the lowest AE values for DS11, DS12, DS13, and DS14 for end-point predictions and next-step predictions. Except for DS13, the AE values of the DNN-RED are consistent with the AB values for DS11, DS12, DS13, and DS14 in terms of next-step prediction. Table 3 shows that the DNN-RED ranks first among all the models for DS11, DS12, DS13, and DS14.

(4) Table 4 shows that the DNN-RED has the lowest R_m value among all the models for 14 fault data sets and ranks first. Its R_m value is 0.07 and 0.94 in terms of end-point predictions and next-step predictions respectively. Ranking second is JordanNet-Generalization and JordanNet-Prediction in terms of end-point predictions and next-step predictions respectively. Their R_m values are 4.75 and 4.46 respectively. Ranking last is the Power

Table 2
Model comparison results for DS6–DS10.

Dataset	Models	End-point predictions			Next-step predictions		
		AE	RANK	AB	AE	RANK	AB
6	FFN-generalization	2.62	3	+1.03	4.33	6	−0.09
	FFN-prediction	2.40	2	−1.78	4.51	7	−2.98
	JordanNet-generalization	3.19	4	−1.56	5.25	9	−0.31
	JordanNet-prediction	5.19	5	−5.19	5.24	8	−4.66
	Logarithmic	7.74	6	+5.04	3.47	2	+0.42
	Inverse polynomial	8.09	7	+1.54	3.59	4	+01.3
	Exponential	8.51	8	+2.68	3.51	3	+0.23
	Power	11.88	9	+11.13	3.51	3	+0.13
	Delayed S-shape	18.92	10	−16.21	3.63	5	−1.96
	DNN-RED	0.08	1	+0.08	0.31	1	+0.19
7	FFN-generalization	6.43	4	+6.28	3.66	9	+1.91
	FFN-prediction	6.80	5	+6.80	3.74	10	+1.58
	JordanNet-generalization	2.27	3	−1.40	2.97	6	−0.50
	JordanNet-prediction	1.31	2	−0.97	2.90	4	−1.61
	Logarithmic	13.20	7	+11.49	2.88	3	+0.52
	Inverse polynomial	13.10	6	+8.3	2.92	5	+0.29
	Exponential	14.56	8	+3.55	2.50	2	−0.03
	Power	24.71	10	+24.71	3.19	8	+1.35
	Delayed S-shape	17.76	9	−16.86	3.07	7	−2.61
	DNN-RED	0.07	1	+0.07	1.21	1	+1.21
8	FFN-generalization	6.83	5	+6.83	6.56	8	+6.51
	FFN-prediction	5.46	4	+5.46	5.24	7	+3.37
	JordanNet-generalization	4.30	2	−0.68	9.11	10	+4.92
	JordanNet-prediction	4.71	3	−3.88	3.73	3	−1.27
	Logarithmic	23.97	8	+22.25	4.47	6	+2.20
	Inverse polynomial	21.08	7	+19.70	4.45	5	+2.14
	Exponential	24.84	9	+16.50	4.26	4	+1.64
	Power	31.08	10	+30.66	7.06	9	+4.99
	Delayed S-shape	11.36	6	−8.66	3.39	2	−0.20
	DNN-RED	0.02	1	+0.02	1.30	1	+1.30
9	FFN-generalization	13.90	5	−13.90	11.82	9	−6.42
	FFN-prediction	12.71	3	−12.71	6.72	3	−2.21
	JordanNet-generalization	13.73	4	−13.73	9.72	5	−5.86
	JordanNet-prediction	11.26	2	−11.26	6.41	2	+0.98
	Logarithmic	19.12	7	−15.15	10.20	6	−5.16
	Inverse polynomial	18.27	6	−9.31	9.09	4	−4.78
	Exponential	30.54	8	−28.67	13.06	10	−12.0
	Power	53.75	10	+34.88	11.36	8	+0.88
	Delayed S-shape	32.98	9	+4.45	10.86	7	−2.26
	DNN-RED	0.05	1	+0.05	1.85	1	+1.85
10	FFN-generalization	8.32	4	+8.32	5.90	8	+3.96
	FFN-prediction	8.80	5	+8.80	7.45	10	+3.65
	JordanNet-generalization	2.41	2	+0.33	4.08	5	+1.45
	JordanNet-prediction	3.01	3	−2.13	3.22	2	−2.34
	Logarithmic	17.00	7	+17.00	3.75	4	+0.69
	Inverse polynomial	28.87	8	+28.37	4.99	6	+2.01
	Exponential	9.15	6	+6.71	3.28	3	−0.17
	Power	42.04	10	+42.04	3.53	7	+2.78
	Delayed S-shape	30.41	9	−30.38	6.57	9	−6.55
	DNN-RED	0.10	1	+0.10	0.32	1	+0.32

model in terms of end-point predictions and next-step predictions. Its R values are 26.42 and 7.44 respectively.

The R_m value of DNN-RED is far lower than average (12.52) in terms of end-point predictions. The R_m value of DNN-RED in terms of next-point predictions is also lower than the average value (5.20). The R_m value of DNN-RED in terms of end-point predictions is lower than that of DNN-RED in terms of next-step predictions. These results illustrate that the DNN-RED can better adapt to end-point predictions than next-step predictions.

Fig. 6 shows the prediction results of the DNN-RED in terms of end-point predictions and next-step predictions utilizing DS1, DS2, DS3, DS4, DS5, DS6, DS7, and DS8. Table 4 indicates that the Std value in terms of end-point predictions is 9.31, which is higher than the Std value of 1.71 in terms of next-step predictions. This result illustrates that the R_m values of all the models are relatively scattered in terms of end-point predictions, whereas the R_m values of all the models are relatively concentrated except for the DNN-RED.

Table 3
Model comparison results for DS11–DS14.

Data set	Models	End-point predictions			Next-step predictions		
		AE	RANK	AB	AE	RANK	AB
11	FFN-generalization	10.59	5	−5.91	7.04	10	+1.03
	FFN-prediction	6.66	4	−2.01	5.23	4	+2.31
	JordanNet-generalization	5.93	2	−3.91	3.62	3	−1.23
	JordanNet-prediction	6.04	3	−5.85	2.97	2	−2.69
	Logarithmic	20.82	7	+4.72	5.74	6	−0.79
	Inverse polynomial	17.75	6	−2.36	5.63	5	−1.19
	Exponential	26.60	8	−5.22	6.02	7	−1.30
	Power	42.52	10	+41.64	6.64	9	+1.12
	Delayed S-shape	35.78	9	−35.78	6.06	8	−3.42
	DNN-RED	0.04	1	+0.04	1.88	1	+1.88
12	FFN-generalization	7.59	3	−4.38	6.36	7	−1.88
	FFN-prediction	8.04	5	−4.57	6.26	6	−2.01
	JordanNet-generalization	6.69	2	−5.28	3.59	2	−1.75
	JordanNet-prediction	7.62	4	−7.30	3.64	3	−3.38
	Logarithmic	12.96	6	−9.06	4.84	4	−3.02
	Inverse polynomial	24.29	9	+7.25	7.67	9	+1.14
	Exponential	17.93	7	−14.57	5.01	5	−3.35
	Power	23.15	8	+18.96	7.96	10	+2.25
	Delayed S-shape	33.78	10	−33.78	6.61	8	−2.29
	DNN-RED	0.02	1	+0.02	1.58	1	+1.58
13	FFN-generalization	5.69	5	−2.72	5.40	10	+2.31
	FFN-prediction	4.88	4	−1.52	4.76	9	+1.98
	JordanNet-generalization	4.24	3	−2.53	4.33	7	+1.17
	JordanNet-prediction	2.54	2	−2.48	2.28	2	−0.63
	Logarithmic	16.33	6	−16.33	3.20	4	−1.23
	Inverse polynomial	17.69	8	−17.69	3.12	3	−1.70
	Exponential	28.87	9	−28.87	3.50	5	−2.15
	Power	17.41	7	+7.49	3.74	6	+0.73
	Delayed S-shape	37.16	10	−37.16	4.70	8	−4.60
	DNN-RED	0.02	1	+0.02	2.14	1	+1.96
14	FFN-generalization	3.36	4	+3.5	4.56	8	+2.76
	FFN-prediction	2.92	2	+1.56	4.80	9	+0.55
	JordanNet-generalization	3.47	5	−2.93	4.86	10	−1.22
	JordanNet-prediction	3.26	3	−3.04	4.50	7	−1.44
	Logarithmic	11.61	7	+3.32	3.24	4	−0.35
	Inverse polynomial	7.88	6	−0.68	3.13	2	−0.59
	Exponential	12.85	8	−1.98	3.52	5	−0.80
	Power	17.66	9	+15.38	3.20	3	+0.59
	Delayed S-shape	19.78	10	−17.10	3.55	6	−1.98
	DNN-RED	0.09	1	+0.09	0.80	1	+0.80

Table 4
Mean of AE measure and rank.

Model	End-point predictions		Next-step predictions	
	Mean of AE	RANK	Mean of AE	RANK
FFN-generalization	6.67	5	6.62	9
FFN-prediction	6.12	4	5.38	5
Jordan Net-generalization	4.75	2	5.47	6
Jordan Net-prediction	4.94	3	4.46	2
Logarithmic	14.23	6	5.04	3
Inverse polynomial	17.93	7	5.64	7
Exponential	18.45	8	5.19	4
Power	26.42	10	7.44	10
Delayed S-shaped	25.61	9	5.80	8
DNN-RED	0.07	1	0.94	1
Average	12.52		5.20	
Std	9.31		1.71	

In summary, we can draw the following points based on the aforementioned experimental results:

(1) The proposed model has the lowest prediction error among all the models utilizing 14 fault data sets in terms of end-point predictions and next-step predictions.

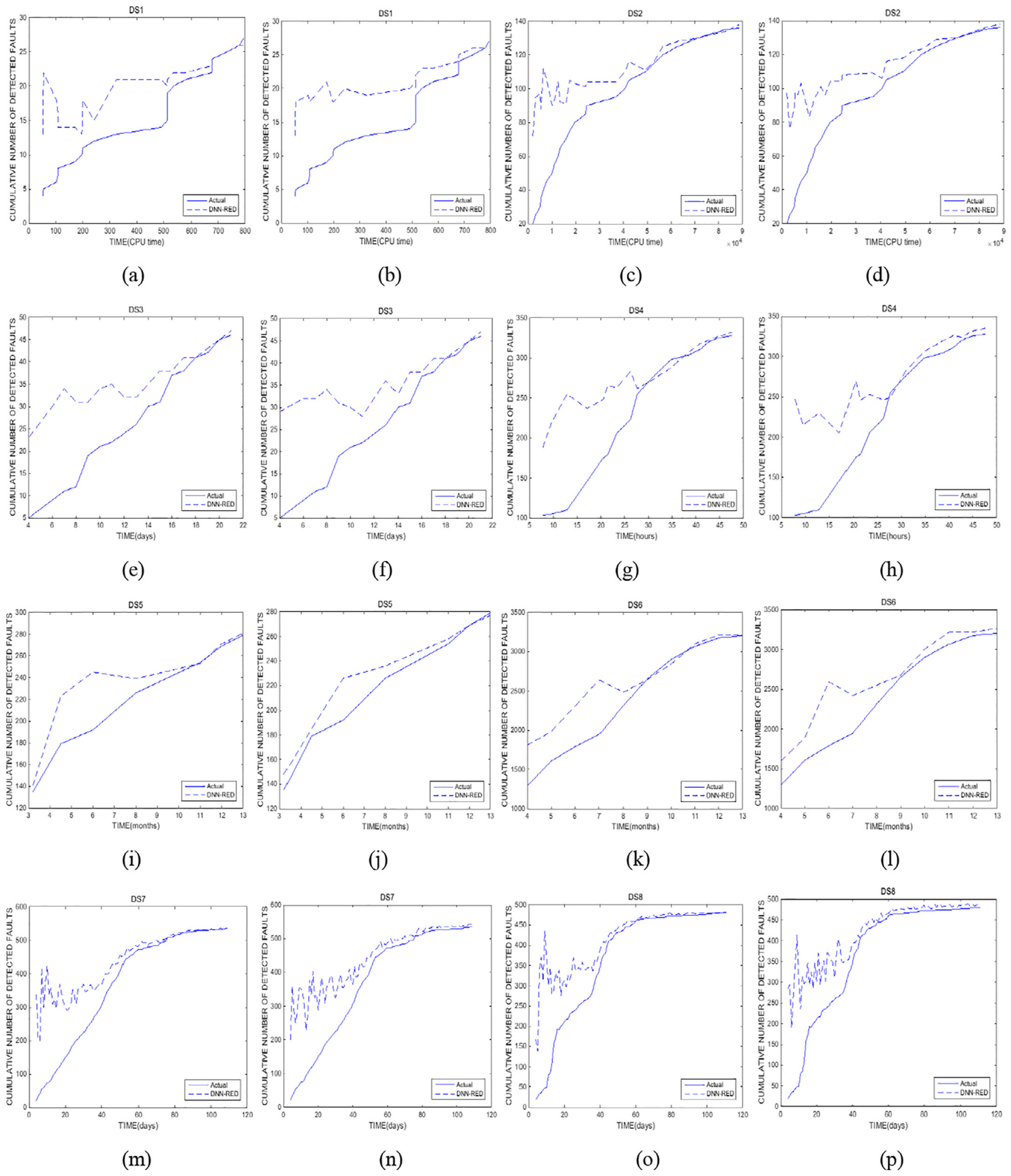

Fig. 6. Plots of the DNN–RED changes and actual observed fault changes over time. (a), (c), (e), (g), (i), (k), (m), and (o) represent the end-point predictions of DNN–RED for DS1, DS2, DS3, DS4, DS5, DS6, DS7, and DS8, respectively. (b), (d), (f), (h), (j), (l), (n), and (p) denote the next-step predictions of DNN–RED for DS1, DS2, DS3, DS4, DS5, DS6, DS7, and DS8, respectively.

(2) The proposed model has better consistency than the four NN models and five parameter models in end-point predictions and next-step predictions.
(3) The proposed model that ranked first among all the models utilizing 14 fault data sets has better prediction performance than the four NN models and five parameter models.
(4) Given that the proposed model can automatically capture the characteristics of the detection faults in the software testing process, it therefore has better adaptability, stability, and accuracy on software fault predictions.

5. Threats to validity

First, the proposed model has underfitting problems because of the limitations in fault data size in the software testing process. The proposed model must also generate underfitting problems due to the encoding and decoding processes in train.

Second, we only adopt the fault data sets from closed-source software projects and do not consider the fault data sets in open-source software projects. Thus, utilizing the proposed deep NN model to evaluate the software reliability of open-source software projects has yet to be implemented.

Finally, more types and number of fault data sets must generally be applied to evaluate and effectively validate the performance of software reliability models. However, comprehensively evaluating software reliability models of different types and with a large number of fault data sets is difficult and unrealistic. Therefore, adopting the suitable types and number of fault data sets to assess the performance of software reliability models remains an unresolved issue.

6. Conclusions

This study compares a deep NN model based on the RNN encoder–decoder with four NN models and five parameter models. The experimental results show that the proposed model has the best prediction performance among all the models in this study for 14 fault data sets. The AE values of the proposed model have the lowest values in terms of end-point predictions and next-step predictions among all the models. Specifically, the proposed model has the lowest prediction error among all the models utilizing 14 fault data sets. Moreover, the proposed model has better consistency than four NN models and five parameter models utilizing 14 fault data sets in terms of end-point predictions and next-step predictions.

Although the NN models have lower prediction errors than the other models reported in [11], the proposed deep NN model has lower prediction errors than NN models. Tables 1–3 show that AE values of the deep model are far lower than those of the four NN models and five parameter models. Thus, the proposed deep NN model proposed can better adapt to software reliability prediction than the other models. The proposed deep NN model has better stability, robustness, and accuracy for software reliability prediction as well.

The prediction of faults for the proposed model was limited by types and quantities of fault data sets used in this paper. In general, the more types and quantities of fault data sets, the better the performance of the model can be evaluated. However, in the software testing process, most of the actual collected fault datasets for software reliability did not label or mark the detailed information on fault properties, such as simple and complicated, dependent and independent faults or introduced and unintroduced faults due to simplicity of faults collected and recorded for testers or debuggers. Thus, we will collect more types and quantities of fault data sets for future research.

In addition, the delay between the fault detection and fault correction is also an important factor in the software testing process. In this paper, we only use the fault detection data as training data for deep NN. In future, we will study how to combine the fault detection data and fault correction data as training data for deep NN. In future research, we will also consider using wider class of artificial neural networks to build software reliability model, such as deep belief networks and convolutional neural networks, etc.

Acknowledgments

We would like to thank the providers for their DeepLearnToolbox tools. We also thank our anonymous reviewers for their valuable comments and suggestions.

References

[1] Musa JD. Software reliability-measurement, prediction, applications. New York: McGraw-Hill; 1987.
[2] Goel AL. Software reliability models: assumptions, limitations and applicability. IEEE Trans Software Eng 1985;SE-11:1411–23.
[3] Wang J, Wu Z. Study of the nonlinear imperfect software debugging model. Reliab Eng Syst Saf 2016;153:180–92.
[4] Crow LH. Reliability for complex repairable systems. Reliability and biometry. SIAM; 1974.
[5] Yamada S, Ohba M, Osaki S. S-shaped reliability growth modeling for software error detection. IEEE Trans Reliab 1983;R-32:475–84.
[6] Littlewood B, Verrall JL. A Bayesian reliability model with a stochastically monotone failure rate. IEEE Trans Reliab 1974;R-23:108–14.
[7] Wang J, Wu Z, Shu Y, Zhang Z. A general imperfect software debugging model considering the nonlinear process of fault introduction. In: Proceedings of the 14th international conference on quality software (QSIC); 2014. p. 222–7.
[8] Okamura H, Dohi T, Osaki S. Software reliability growth models with normal failure time distributions. Reliab Eng Syst Saf 2013;116(8):135–41.
[9] Wang J, Wu Z, Shu Y, Zhang Z. An imperfect software debugging model considering log-logistic distribution fault content function. J Syst Software 2015;100:167–81.
[10] Karunanithi N, et al. Prediction of software reliability using neural networks. Proc 1991 IEEE lnt Symp Software Reliab Eng May 1991:124–30.
[11] Karunanithi N, Whitley D, Malaiya YK. Prediction of software reliability using connectionist models. IEEE Trans Software Eng 1992;18(7):563–74.
[12] Karunanithi N, Whitley D, Malaiya YK. Using neural networks in reliability prediction. IEEE Software 1992;9(4):53–9.
[13] Wang J, Wu Z, Shu Y, Zhang Z, Xue L. A study on software reliability prediction based on triple exponential smoothing method. In: In proceedings of the 46th summer computer simulation conference (SCSC 2014), 46; 2014. p. 440–8.
[14] Whitley D, Karunanithi N. Improving generalization in feed-forward neural networks. In: Proc IJCNN, 11; July 1991. p. 77–82.
[15] Hinton GE, Salakhutdinov RR. Reducing the dimensionality of data with neural networks. Science 2006;313(5786):504–7.
[16] Hinton GE, Osindero S, Teh YW. A fast learning algorithm for deep belief nets. Neural Comput 2006;18(7):1527–54.
[17] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 2014;15(1):1929–58.
[18] Graves A. Generating sequences with recurrent neural networks. Comput Science 2013.
[19] Sutskever H, Vinyals O, Le Q. Sequence to sequence learning with neural networks. In: Advances in neural information processing systems (NIPS 2014), 4; 2014. p. 3104–12.
[20] Pascanu R, Gulcehre C, Cho K, Bengio Y. How to construct deep recurrent neural networks. In: Proceedings of the second international conference on learning representations (ICLR 2014), April; 2014.
[21] Cho K, Merrienboer BV, Gulcehre C, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the empiricial methods in natural language processing (EMNLP 2014); 2014.
[22] Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: International conference on software engineering. ACM; 2016. p. 297–308.
[23] Hu QP, Xie M, Ng SH, et al. Robust recurrent neural network modeling for software fault detection and correction prediction. Reliab Eng Syst Saf 2007;92(3):332–40.
[24] Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: International conference on software engineering. IEEE; 2017. p. 297–308.
[25] Tian L, Noore A. Evolutionary neural network modeling for software cumulative failure time prediction. Reliab Eng Syst Saf 2005;87(1):45–51.
[26] Bai CG, Hu QP, Xie M, et al. Software failure prediction based on a Markov Bayesian network model. J Syst Software 2005;74(3):275–82.
[27] Costa EO, Souza GAD, Pozo ATR, et al. Exploring genetic programming and boosting techniques to model software reliability. IEEE Trans Reliab 2007;56(3):422–34.
[28] Li H, Zeng M, Lu M, et al. Adaboosting-based dynamic weighted combination of software reliability growth models. Quality Reliab Eng Int 2012;28(1):67–84.
[29] Torrado N, Wiper MP, Lillo RE. Software reliability modeling with software metrics data via Gaussian processes. IEEE Trans Software Eng 2013;39(8):1179–86.
[30] zhao W, TTao ZSDing, et al. A dynamic particle filter-support vector regression method for reliability prediction. Reliab Eng Syst Saf 2013;119:109–16.
[31] Roy P, Mahapatra GS, Dey KN. Neuro-genetic approach on logistic model based software reliability prediction. Expert Syst Appl 2015;42(10):4709–18.
[32] Matsumoto K, et al. Experimental evaluation of software reliability growth models. In: Proc IEEE Conf FTCS-18; June 1988. p. 148–53.

[33] Ohba M. Software reliability analysis models. lBM J Res Develop July 1984;28:428443.
[34] Shooman ML. Probabilistic models for software reliability prediction. In: Statistical computer performance evaluation. Academic; 1972. p. 485–502.
[35] Tohma Y, et al. Parameter estimation of the hyper-geometric distribution model for real test/debug data. Dept Computer Science, Tokyo Inst Tech; 1990. Tech Rep 901002.
[36] Tohma Y, et al. Structural approach to the estimation of the number of residual software faults based on the hyper-geometric distribution. IEEE Trans Software Eng Mar. 1989;15:345–55.
[37] Anna-Mary BM. A study of the Musa reliability model. Univ. Maryland; 1980.
[38] Malaiya YK, Karunanithi N, Verma P. Predictability measures for software reliability models. In: Proc. 14th IEEE lni. computer software and applications conference; Oct. 1990. p. 7–12.
[39] Brocklehurst S, et al. Recalibrating software reliability models. IEEE Trans Software Eng Apr. 1990;16:458–70.