2018 IEEE International Conference on Systems, Man, and Cybernetics

Bug Detection based on LSTM Networks and Solution Codes

Yunosuke Teshima
University of Aizu, Japan
Email: m5221151@u-aizu.ac.jp

Yutaka Watanobe
University of Aizu, Japan
Email: yutaka@u-aizu.ac.jp

***Abstract*—Debugging a program is always an obstacle to programmers and learners. In particular, novice programmers waste a lot of time finding bugs, so a feedback system to support debugging is required. Although existing editors and IDEs support finding syntax errors, their functions for detecting logical errors are limited. In the present paper, we present bug detection methods for the feedback system of an online judge system which contains many programming problems and accumulates numerous lines of solution source code. The proposed method uses the solutions and a language model based on long short-term memory (LSTM) networks for bug detection. In addition, since LSTM networks have some hyperparameters, we investigate the best model for bug detection in terms of perplexity and training time. The results of experiments show that models trained by solutions can detect bugs in a compiled code based on the static structure of a program.**

I. Introduction

Along with the rapid growth of artificial intelligence (AI), a number of new initiatives in various fields, such as industry, healthcare, and other business, have been promoted by various companies, organizations, and universities. Educational fields in which students and teachers should make decisions for learning using various tools also constitute an application area of AI. Although the possibilities of information technologies and data mining have been widely researched [1], the practical use and effects of AI in education remain limited to traditional functions such as recommendation systems. In particular, programming has been growing rapidly in both academia and industry. Accordingly, numerous teaching materials and different types of online services collaborating with interactive interfaces, social media, and gamification, for example, are available now to support the learning process. Although these materials are attractive, in order to maintain motivation in programming education, debugging should be supported because programming remains a very difficult task requiring logical thinking, and syntactical and conceptual errors are frequently involved. As such, awareness computing, which can support decision making based on activities and solutions of a number of users on a common learning platform, is a promising approach.

The online judge (OJ) system [2] is an e-learning system which provides a set of problems and judges submitted programs. The system automatically compiles and runs a submitted source code and tests whether the code is correct using the corresponding judge data. An OJ was originally developed for programming contests, but OJs are very useful for programming education. When using an OJ, instructors do not need to prepare materials including problems and the corresponding data and to score submitted source codes.

Although automatic judges are very useful, generally the verdict includes only the state of correctness, the CPU time, and the used memory size, as well as error messages from compilers. As such, the learner cannot recognize his/her mistakes and wastes a lot of time debugging his/her program. In particular, logical errors are difficult to find. So, a promising approach is to provide a feedback system for learners debugging their programs.

In the present paper, we propose bug detection methods based on a language model [3], which is a kind of probability distribution, and long short-term memory (LSTM) networks [4]. In addition, since LSTM networks have some hyperparameters, we investigate the best model for practical bug detection. The primary goal of the present study is to develop a feedback system which highlights the possibilities of bugs in program lines so that the learner can find bugs within a limited section of the program code. Implementation and evaluation are conducted using data from the Aizu Online Judge (AOJ) [5] [6], which is a commonly used OJ [1]. Thus far, approximately 50,000 users have registered to use the AOJ, and several millions of source codes have been accumulated. We train the language model using the accumulated solutions. By deep learning, the model can calculate the appearance probabilities of words (program elements) based on the static structure of a program. We demonstrate how bug detection of the model works using incorrect-answer source codes accumulated in the AOJ.

The remainder of the present paper is organized as follows. Related research is presented in section 2. The model architecture is described in section 3. The methods of training, evaluation, and bug detection are presented in section 4. Section 5 presents the results and provides a discussion. Finally, section 6 concludes the paper.

II. Related Research

A syntax error highlighter and other means of support, such as supplement and suggestion, implemented in advanced editors are indispensable functions in writing a program. Although developers and researchers of modern IDEs and their plug-ins actively consider such advanced functions, we should

2577-1655/18/$31.00 ©2018 IEEE
DOI 10.1109/SMC.2018.00599

mention approaches other than traditional static program analysis to find errors and other useful features in a program.

Some methods for detecting, finding, and predicting bugs in source codes have already been proposed by various organizations. For example, [7] considered bug detection based on machine learning and a large corpus of programs. However, the results are limited, and bug finding based on machine learning is inferior to static program analysis. Another example is the use of a support vector machine for code clone detection [8]. Although machine learning approaches are attractive, the accuracy of the results depend on the experimental process and the size of data, as well as on how future vectors are created from source codes. Since a program code is a stream of operations, the order of elements in the static structure must be considered in order to define the corresponding future vectors.

The present research is inspired by entropy based bug prediction using neural network based regression, as well as bug detection using an n-gram language model [9] [10]. The use of an n-gram language model for bug detection was proposed, and detection of an n-gram language model was found to be superior to static program analysis [10]. In addition, bug prediction based on statistical linear regression and neural network based regression were compared [9]. Although the detection methods proposed by [9] and [10] can detect most types of bug, they can not detect algorithm errors.

III. Architecture of the Model

A. Long Short-Term Memory Networks

An LSTM network is a special kind of recurrent neural network which prevents the vanishing/exploding gradient problem [4]. An LSTM network outputs a vector based on one input datum and the results are calculated in advance. This means that an LSTM network can process an arbitrary sequence of data. Fig. 1 shows the proposed LSTM network for bug detection, which is constructed as follows:

- Embedding layers as input layers,
- Using the softmax function as an activation function of output layers,
- Using dropout [11] for regularization, and
- Using Adam [12] as an optimizer.

The embedding layer is a layer which receives an ID and maps the ID to a vector. The softmax function is an activation function to convert output into probability. The softmax function receives a vector $\mathbf{x} = [x_1, \cdots, x_n]$ and returns a vector $\mathbf{p} = [p_1, \cdots, p_n]$, which denotes the probabilities of each element. The softmax function calculates $\mathbf{p}$ as follows:

$$p_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \qquad \text{for } i = 1, \cdots, n.$$

An LSTM network that is constructed by the above process can calculate the appearance probabilities of each element based on a sequence of IDs. We use a language model which calculates probabilities based on an LSTM network whose IDs represent words (elements) in a program.

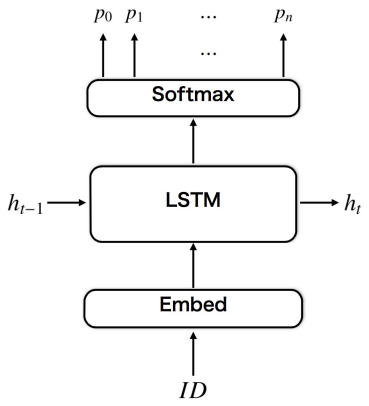

Fig. 1: Proposed LSTM network for bug detection

B. Hyperparameters

We regularize LSTM networks by dropout. The dropout ratio is 0.5, as recommended by [11]. We optimize LSTM networks by Adam, which has four hyperparameters. We select $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e-8$ based on the recommendation of [12].

In the proposed method, a program is divided into 157 different types of words. Thus, the number of units in the input and output layers is 157. The number is decided by the number of elements. On the other hand, it is necessary to find and decide the number of units in the hidden layers for the optimal solution. We prepare five models which have 100, 200, 300, 400, and 500 units in hidden layers. The model with 100 units in hidden layers is referred to as the 100-unit model, and the other models are referred to in a similar manner. We train and evaluate these models, and select the best model for bug detection.

IV. Experiment using the Online Judge System

A. Materials

We focus on a problem related to an insertion sort algorithm which sorts a sequence of integers in ascending order (AOJ: ALDS1_1_A [13]). An integer N which indicates the number of elements in the sequence is given in the first line, and N elements of the sequence are then separated by a single space in the second line. The output consists of N lines. The sequence should be printed in a line for each step of the insertion sort algorithm. Elements of the sequence should be separated by a single space. Fig. 2 shows an example of input and output data of ALDS1_1_A. More than 15,000 submitted source codes written in different programming languages were examined in this problem. In the experiment, we used 1,132 accepted solution source codes written in the C language.

B. Source Code Conversion

It is difficult to train and evaluate models using the raw data of a source code. Before training, evaluation, and bug detection, source codes must each be divided into a sequence of words. In addition, we must encode each word into an ID.

Sample Input 1

```
6
5 2 4 6 1 3
```

Sample Output 1

```
5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6
```

Fig. 2: Example of input and output data for problem ALDS1_1_A

Comments are not necessary for bug detection, so we must first delete all comments. Moreover, we must delete feed lines (\n), tabs (\t), and spaces that are not surrounded by single or double quotation marks. As the dividing method, we divide a source code into a sequence of function names, variable names, keywords, and characters. In the present study, function names, variable names, keywords, and characters are referred to as words. After dividing the source code, we encode each word into an ID. Function names are encoded into IDs 0 to 9 in order of appearance. Variable names are encoded into IDs 10 to 29 in order of appearance. Keywords and characters are encoded into IDs according to Table 1.

Fig. 3 shows an example of the dividing and encoding processes. The sample source code has three functions and three variables. The three functions square, main, and printf are encoded into IDs 0 to 2. The three variables num, var, and array are encoded into IDs 10 to 12. Finally, keywords and characters contained in the sample source code are encoded into IDs according to Table 1.

C. Training and Evaluation

We divide 1,132 solutions into 1,122 training data and 10 evaluation data. We train each model using the 1,122 training data for 50 epochs and calculate the average perplexity of the 10 evaluation data for each epoch. Perplexity is a measurement of how well a probability model predicts a sample. The domain of the perplexity is $[1, +\infty]$. As the value of perplexity becomes lower, the model becomes better. The perplexity is defined by:

$$2^H \text{where: } H = \frac{1}{|D|}\sum_{i=1}^{|D|} -\log_2(P(w_i))$$

where

- $|D|$: length of a sample
- w_i : ID in a sample
- $P(w_i)$: probability of w_i which a model calculates

In addition, we measure the training time for each model and decide the model for bug detection considering the perplexities and training time.

TABLE I: Keywords and characters encoded into IDs

ID	word	ID	word	ID	word	ID	word
30	auto	62		94	>	126	`
31	break	63	!	95	@	127	a
32	case	64	?	96	A	128	b
33	char	65	_	97	B	129	c
34	const	66	"	98	C	130	d
35	continue	67	#	99	D	131	e
36	default	68	$	100	E	132	f
37	do	69	%	101	F	133	g
38	double	70	&	102	G	134	h
39	else	71	'	103	H	135	i
40	enum	72	(	104	I	136	j
41	exturn	73	)	105	J	137	k
42	float	74	*	106	K	138	l
43	for	75	+	107	L	139	m
44	goto	76	,	108	M	140	n
45	if	77	-	109	N	141	o
46	int	78	.	110	O	142	p
47	long	79	/	111	P	143	q
48	register	80	0	112	Q	144	r
49	return	81	1	113	R	145	s
50	short	82	2	114	S	146	t
51	signed	83	3	115	T	147	u
52	sizeof	84	4	116	U	148	v
53	static	85	5	117	V	149	w
54	struct	86	6	118	W	150	x
55	switch	87	7	119	X	151	y
56	typedef	88	8	120	Y	152	z
57	union	89	9	121	Z	153	{
58	unsigned	90	:	122	[	154	|
59	void	91	;	123	\	155	}
60	volatile	92	<	124	]	156	~
61	while	93	=	125	^		

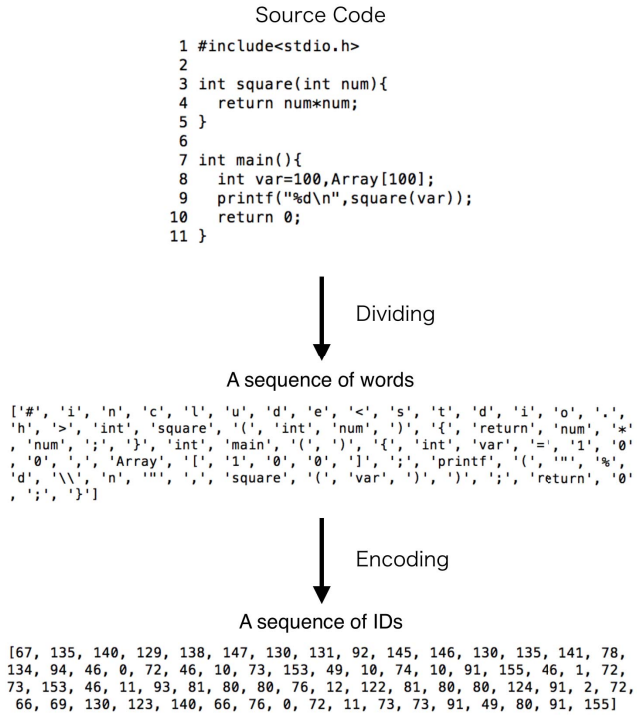

Fig. 3: An example of dividing and encoding processes

D. Bug Detection

Models are trained by solution source codes which have been accepted by the judge. This means that words with low probability calculated by the models can be considered to be unnatural in the flow of a program for the problem. We define words whose probabilities are less than 0.1 as bug candidates. We detect all of the candidates in a source code based on the calculation of the model which we selected. On the other hand, the online judge detected numerous incorrect-answer source codes for ALDS1_1_A. Among these source codes, we selected three examples of incorrect-answer source codes with different characteristics and demonstrate how bug detection of the model works using these codes.

V. Results

A. Perplexities and Training Time

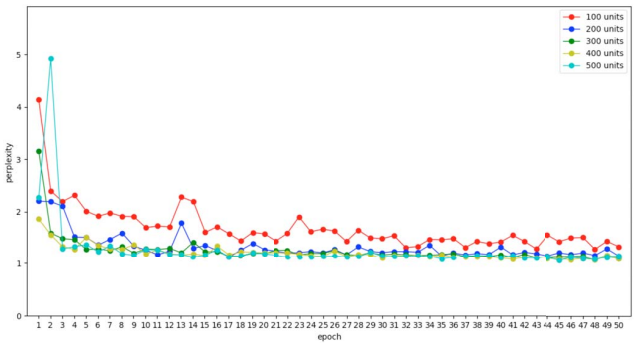

Fig. 4: Perplexities for each epoch and each model

TABLE II: Training time and perplexity of the last epoch for each model

Number of units in hidden layer	Time (sec)	Perplexity at the last epoch
100	30834.167	1.32573686078
200	46388.059	1.14466136435
300	56910.655	1.10886346477
400	75502.111	1.10991073126
500	104504.209	1.14536090668

Fig. 4 shows the result of training and evaluation. The X-axis indicates the epoch, and the Y-axis indicates the perplexity. In Fig. 4, the 100- and 200-unit models yield worse results than the other models at some epochs. On the other hand, the 300-, 400-, and 500-unit models yield approximately the same results. Table 2 shows the training time and perplexity at the last epoch. The 200-, 300-, 400-, and 500-unit models have approximately the same perplexity at the last epoch. If we select a model for bug detection by considering only Table 2, then the 200-unit model is selected. However, the results for the 200-unit model exhibit slight up-and-down swings, as shown in Fig. 4. This means that the model is not sufficient to express a program for the problem. Therefore, we decided to use the 300-unit model for bug detection.

B. Bug Detection

Fig. 5, 6, and 7 show incorrect-answer source codes with line number and underlines, which indicate the possibility of bugs detected by the 300-unit model.

Tables 3, 4, and 5 show words whose probabilities are less than 0.1 in the source codes shown in Fig. 5, 6, and 7, respectively. In these tables, words whose appearence probability is heighest for each detection are also provided. The words can be considered to be correct words which the 300-unit model predicted. Note that variables and functions are surrounded by single quotation marks.

In Fig. 5, line 19 of the source code contains a bug whereby a variable i is assigned a value of 0 but should be assigned a value of 1.

In Fig. 6, this source code has no function to print \n after line 14 and before line 16. This is a bug in the source code.

The source code in Fig. 7 simply prints the sequence n-1 times. However, the sequence should be printed n times. It is necessary to print the sequence after line 10 and before line 12. In addition, the printing format is incorrect, because a space should not be printed after the last element. These two items are bugs in the source code.

```
 1 #include<stdio.h>
 2 #define N 100
 3
 4
 5 int main(){
 6   int n,A[N+1],k;
 7   int i;
 8   scanf("%d",&n);
 9   for(i=0;i<n;i++){
10     scanf("%d",&A[i]);
11   }
12   for(k=0;k<n;k++){
13     if(k>0)printf(" ");
14     printf("%d",A[k]);
15   }
16   printf("\n");
17
18 int j,v;
19 for(i=0;i<n;i++){
20   v=A[i];
21   j=i-1;
22   while(j>=0 && A[j]>v){
23     A[j+1]=A[j];
24     j--;
25     A[j+1]=v;
26   }
27   for(k=0;k<n;k++){
28     if(k>0)printf(" ");
29     printf("%d",A[k]);
30   }
31   printf("\n");
32  }
33 return 0;
34 }
```

Fig. 5: Incorrect-answer source code with underlines

TABLE III: Possibilities of bugs predicted by the 300-unit model for the source code in Fig. 5

Word	Line number	Probablity	Predicted word
+	6	0.0031167	]
int	18	0.0262439	for
'j'	18	0.0899055	'i'
'i'	19	0.0802422	'k'
0	19	0.0076889	1
'v'	22	0.0685058	'i'
'v'	25	0.0759895	'i'

```
 1 #include<stdio.h>
 2 int main(){
 3       int i,j,N;
 4       int A[100];
 5       int v;
 6
 7       scanf("%d",&N);
 8       for(i=0;i<N;i++){
 9               scanf("%d",&A[i]);
10       }
11       for(j=0;j<N;j++){
12               printf("%d",A[j]);
13               if(j!=N-1)printf(" ");
14       }
15
16       for(i=1;i<N;i++){
17               v=A[i];
18               j=i-1;
19               while(j>=0&&A[j]>v){
20                       A[j+1]=A[j];
21                       j--;
22               }
23               A[j+1]=v;
24
25               for(j=0;j<N;j++){
26                       printf("%d",A[j]);
27                       if(j!=N-1)printf(" ");
28               }
29               printf("\n");
30       }
31       return 0;
32 }
33
```

Fig. 6: Incorrect-answer source code with underlines

TABLE IV: Possibilities of bugs predicted by the 300-unit model for the source code in Fig. 6

Word	Line number	Probablity	Predicted word
;	3	0.0920520	,
for	16	0.0004707	'printf'

```
 1 #include <stdio.h>
 2
 3
 4 int main(){
 5     int N,i,j,num[1001],v,x;
 6
 7     scanf("%d",&N);
 8     for (i=0; i<N; i++) {
 9         scanf("%d",&num[i]);
10     }
11
12     for (i=1; i<N; i++) {
13         v=num[i];
14         j=i-1;
15         while (j>=0 && num[j]>v) {
16             num[j+1]=num[j];
17             j--;
18         }
19
20         num[j+1]=v;
21         for (x=0; x<N;x++) {
22           printf("%d ",num[x]);
23         }
24          printf("\n");
25
26     }
27
28     return(0);
29 }
```

Fig. 7: Incorrect-answer source code with underlines

TABLE V: Possibilities of bugs predicted by the 300-unit model for the source code in Fig. 7

Word	Line number	Probablity	Predicted word
1	5	0.0243946	]
'v'	13	0.0067056	for
while	15	0.0217507	for
'x'	21	0.0817247	'j'
[1]	22	0.0000378	"
\	24	0.0000822	%
(	28	0.0032100	0

[1] A space

C. Discussion

In this section we discuss the results of the experiment and point out some limitations of the proposed approach.

The main purpose of the proposed feedback system is to limit the range in which submitters have to search for bugs. So, some underlines which do not include bugs in Fig. 5, 6 and 7 are irrelevant to the feedback system. On the other hand, it is the fatal error that underlines as a whole do not include true bugs.

Table 3 shows possibilities of bugs which the 300-unit model detected in the source code in Fig. 5. The bugs include a true bug, which should be fixed. In addition, the bug in the code is underlined. Therefore, in this case, we successfully detected the bug and limited the area in which the submitter must search for bugs. The important point is that the submitter

need only search five lines for bugs.

Table 4 shows that a 300-unit model predicted that $printf$ should be used in place of for in line 16. Although the prediction is not incorrect, underlines do not appear to indicate the bug in Fig. 6. Although there are no bugs in lines 3 and 16, the submitter is going to look for bugs in line 3 and 16. This example indicates that bug detection using LSTM networks cannot determine the lack of something which a submitter should have included in the code.

Table 5 and Fig. 7 show that the 300-unit model can detect bugs and underlines are drawn clearly regarding the format of printing. However, the 300-unit model does not appear to be able to detect a lacking of printing. In the ALDS1_1_A problem, there are two ways to print the sequence n times. The first is to print the sequence before insertion sorting and as the last of steps of insertion sorting, as in the source codes in Fig. 5 and 6. The second is to print the sequence as the first steps of insertion sorting and again after insertion sorting. The fact is that the 300-unit model can detect a lacking of printing. Actually, Table 5 shows that the 300-unit model predicted that, in line 13, for should be replaced by variable v, which means that the 300-unit model conjectured that the source code in Fig. 7 uses the second printing method. Since the source code prints the sequence as the last steps of insertion sorting, the source code appears to print the sequence using the first method. Thus, although prediction is not incorrect, it is poor for the feedback system. The reason the 300-unit model guessed the second method is that LSTM networks make predictions based only on the previous sequence. The prediction of 300-unit model at first of line 12 does not consider whether the source code is going to print the sequence in the final steps of insertion sorting. Finally, prediction using LSTM networks does not work correctly when the prediction should use a sequence which will appear later in the code.

VI. Conclusion and future research

In the present paper, we proposed bug detection based on solutions and a language model using LSTM networks as a feedback system. In addition, we investigated the best model for bug detection. A model having 300 units in hidden layers provided the best results in terms of perplexity and training time. We demonstrated how bug detection of the selected model worked using three incorrect-answer source codes and evaluated how well bug detection of the model worked. The demonstration revealed that the model can detect most bugs which should be fixed. On the other hand, the detection did not work for some cases. The reasons for its failure to work include that the prediction of the model does not use the parts of the sequence which appear later in the code.

In the future, we plan to use bidirectional long short-term memory (BLSTM) networks which are special kind of bidirectional recurrent neural networks [14] for bug detection. Since source codes are tree structures, BLSTM networks can be expected to have higher accuracy than LSTM networks.

References

[1] A. Dutt, M. A. Ismail, and T. Herawan, "A systematic review on educational data mining," *IEEE Access*, vol. 5, pp. 15 991–16 005, Jan. 2017.

[2] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys*, vol. 1, no. 1, Jan. 2016.

[3] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research 3*, pp. 1137–1155, Feb. 2003.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation 9(8)*, vol. 9, pp. 1735–1780, Nov. 1997.

[5] "Aizu Online Judge," https://onlinejudge.u-aizu.ac.jp/, 2004–2017.

[6] Y. Watanobe, "Development and operation of an online judge system: Aizu online judge," *Information Processing*, vol. 56, no. 10, pp. 998–1005, Oct. 2015.

[7] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs: An initial report," *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*, pp. 21–26, Feb. 2017.

[8] S. Jadon, "Code clones detection using machine learning technique: Support vector machine," *IEEE, International Conference on Computing, Communication and Automation*, pp. 299–303, Apr. 2016.

[9] A. Kaur, K. Kaur, and D. Chopra, "Entropy based bug prediction using neural network based regression," *IEEE, International Conference on Computing, Communication and Automation*, pp. 168–174, May. 2015.

[10] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," *IEEE/ACM International Conference on Automated Software Engineering*, pp. 708–719, Sept. 2016.

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, Jan. 2014.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, pp. 1–13, Dec. 2014.

[13] "ALDS1_1_A," https://onlinejudge.u-aizu.ac.jp/#/courses/lesson/1/ALDS1/1/ALDS1_1_A.

[14] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.