Code completion for programming education based on deep learning

Kenta Terada* and Yutaka Watanobe

Computer Science and Engineering,
University of Aizu,
Fukushima, 965-8580, Japan
Email: m5231117@u-aizu.ac.jp
Email: yutaka@u-aizu.ac.jp
*Corresponding author

Abstract: In solving programming problems, it is difficult for beginners to create a program from scratch. One way to navigate this difficulty is to suggest the next word following an incomplete program. In the present study, we propose a method for code completion characterised by two principal elements: the prediction of the next within-vocabulary word and the prediction of the next referenceable identifier. For the prediction of within-vocabulary words, a neural language model based on an LSTM network with an attention mechanism is proposed. Additionally, for the prediction of referenceable identifiers, a model based on a pointer network to a given incomplete program is proposed. For evaluation of the proposed method, source code accumulated in an online judge system is used. The results of the experiment demonstrate that in both statically and dynamically typed languages, the proposed method can predict the next word to a high degree of accuracy.

Keywords: programming education; code completion; deep learning; LSTM; pointer network.

Reference to this paper should be made as follows: Terada, K. and Watanobe, Y. (2021) ‘Code completion for programming education based on deep learning’, *Int. J. Computational Intelligence Studies*, Vol. 10, Nos. 2/3, pp.78–98.

Biographical notes: Kenta Terada received his BS from the University of Aizu in 2019. He is currently a first-year Master’s student at the University of Aizu. His research interests include programming education based on deep learning.

Yutaka Watanobe received his MS and PhD from the University of Aizu in 2004 and 2007 respectively. He was a Research Fellow of the Japan Society for the Promotion of Science at the University of Aizu in 2007. He is currently a Senior Associate Professor at the University of Aizu. His research interests include data mining, e-learning systems, visual programming and programming education.

This paper is a revised and expanded version of a paper entitled 'Code completion for programming education based on recurrent neural network' presented at 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA), Hiroshima, Japan, 9–10 November 2019.

1 Introduction

With the fourth industrial revolution bringing about various social and industrial changes, the cultivation of human resources with information literacy, such as the ability to use information and communication technology (ICT) for identifying and solving problems, is increasingly required (Chung and Kim, 2016). Accordingly, the number of students who study programming techniques at educational institutions, including high schools and universities, is rapidly increasing. Furthermore, programming is becoming a compulsory subject at elementary schools in many countries. Therefore, the need for accurate and efficient programming education is clearly becoming increasingly important.

One of the simplest ways to encourage the growth of new programming learners is to let the learners solve many problems and become familiar with programming itself. However, many beginners have trouble completing a program from scratch without any hints for solving given problems, which can instead cause a decrease in their learning efficiency. Therefore, it is necessary to help such beginners solve problems without stress caused by cognitive hurdles or thinking for a long time without writing any code. Toward this objective, we focus on providing a function for automatic code completion, which suggests the most ideal candidate for the next word, while learners are writing a program. This function can aid learners by discouraging pausing and spending a lot of time only thinking about the problem. In addition, candidate words suggested by the function encourage learners to come up with a solution to the problem, and learners can make progress in learning programming more swiftly and efficiently.

Currently, there are many educational tools that provide programming problems to learners. One commonly used tool is the online judge (OJ) system, which is an online execution environment that allows learners to tackle many exercises and grades their source code automatically (Wasik et al., 2018). Many OJ systems have been established and are used by students and engineers around the world to acquire programming skills. OJ systems compile and execute user-submitted programs and automatically evaluate their accuracy and performance through various test cases and verification machines. In addition, OJ systems provide a quantitative evaluation of program performance, such as execution time and memory usage.

The Aizu Online Judge (AOJ) system is one of the major OJ systems developed and operated by the University of Aizu (Watanobe, 2018, 2015). Thus far, AOJ contains over 2,000 problems. More than 65,000 users have registered with AOJ, and around four million source codes have been submitted. AOJ contains a variety of problems, such as those pertaining to programming competitions (international collegiate programming contest, high school programming contest, etc.) and those for learning ('introduction to programming', 'algorithms and data structures', etc.). Each problem has limits on CPU time and memory usage, and users can cultivate programming skills by solving problems under their limits. When users submit source code, AOJ immediately provides the users with feedback (accepted/rejected) based on various test cases for the problem, along with the limits. There are eight types of failure: compile error, judge not available, runtime error, time limit exceeded, memory limit exceeded, output limit exceeded, wrong answer, and presentation error. If the program does not fail in terms of any of the eight conditions, then the program is judged as accepted. Users can repeatedly debug and resubmit source code based on the feedback until their program is judged as accepted. At present, however, many OJ systems, including AOJ, do not have functions to help users write programs, such as automatic code completion and bug detection. Introducing these functions into OJ systems would be a great help to many novice programmers.

In the present paper, we propose an algorithm of code completion that predicts the next word when an incomplete program is given. The main situation is assuming a learner is writing a program from scratch to solve a problem, and when he/she wants to know the next word, the proposed algorithm can predict the next plausible word and suggest it to the learner. The next word may be one of two types: a within-vocabulary word or a referenceable identifier. A within-vocabulary word is a word within the vocabulary constructed in pre-processing, including identifier names, reserved words, and operators. An identifier is a user-defined symbol used to uniquely identify a program element. It is possible to select the next referenceable identifier from the code snippet because the identifier was either declared beforehand or was already used for operations such as calculations. Therefore, the prediction of the next referenceable identifier is expected to be improved by selecting the identifier directly from the code snippet instead of the vocabulary. We propose two prediction models and one switching model: a neural language model for predicting the next within-vocabulary word, a pointer model for predicting the next referenceable identifier, and a switching model for determining whether the next word is a referenceable identifier. The neural language model is based on an LSTM network (Hochreiter and Schmidhuber, 1997) with an attention mechanism (Bahdanau et al., 2015; Luong et al., 2015), which is capable of learning long-term dependencies. The attention mechanism solves the problem of forgetting the information relating to a word that has been input long before or failing to capture the dependencies between distant words. This mechanism has contributed to higher accuracy in various tasks of natural language processing, such as machine translation (Jia, 2019), coherent dialogue (Mei et al., 2016), and sentiment analysis (Wang and Liu, 2019). The pointer model is based on a pointer network (Vinyals et al., 2015) to a given incomplete program, which solves the problem of a variable-size vocabulary using a mechanism of neural attention. This model uses attention as a pointer to select a word of the input sequence converted from the incomplete program as the output. The switching model is a binary classification model based on an LSTM network with an attention mechanism, which determines which model to apply in predicting the next word. Experiments and evaluations of the proposed approach are carried out using the AOJ dataset. When considering code completion for a problem, we demonstrate that the proposed models can suggest the next plausible word for solving the problem by learning the code structures of the correct programs that are solutions for the problem. If there are many accepted source codes for problems, then the algorithm is applicable not only to AOJ but also to many other related systems.

The remainder of the present paper is organised as follows. In Section 2, we discuss related research. Sections 3 and 4 describe data pre-processing and the model architecture, respectively. Section 5 evaluates the model and discusses the experimental results. Finally, conclusions in this domain are presented in Section 6.

2 Related research

White et al. (2015) motivated deep learning for software language modelling and showed how a particular deep learning model can remember its state to effectively model sequential data, e.g., streaming software tokens. Their models show that deep learning induces high-quality models compared to n-grams and cache-based n-grams on a corpus of Java projects. Dam et al. (2016) proposed an approach to build a language model for software code upon the powerful deep learning-based LSTM architecture that is capable of learning long-term dependencies which occur frequently in software code. Results from their preliminary evaluation show the effectiveness of LSTM, serving as a concrete indication that LSTM is a promising model for software code.

In order to improve the performance of code completion tasks more, several methods that extend neural language models described above have been proposed. Bhoopchand et al. (2016) introduced a neural language model with a sparse pointer network aimed at capturing very long-range dependencies. By augmenting this model specialised in referring to predefined classes of identifiers, the model obtains a much lower perplexity and a 5% increase in accuracy for code suggestion compared to an LSTM baseline. Ginzberg et al. (2017) presented a deep learning approach to code completion for non-terminals (program structural components) and terminals (program text) that takes advantage of running dependencies to improve predictions. An LSTM model was developed and augmented with several approaches to attention in order to better capture the relative value of the input, hidden state, and context. Li et al. (2017) proposed a pointer mixture network for better predicting out-of-vocabulary (OoV) words in code completion inspired by the prevalence of locally repeated terms in program source code and the recently proposed pointer copy mechanism. Based on context, the pointer mixture network learns to either generate a within-vocabulary word through a recurrent neural network (RNN) component or to regenerate an OoV word from local context through a pointer component. Zhong et al. (2019) described a code suggestion prototype system for JavaScript based on Jupyter Notebook to provide completion for multiple successive code units. Their study has three principal elements: providing a JavaScript pre-processing solution for feature extraction, applying several deep learning technologies to support system performance, and designing a solution for model deployment to provide post-processing methods that improve user experience. All studies described above evaluate the performance of code completion in software engineering by training and evaluating the proposed model using source code collected from GitHub. In this study, we aim to make a contribution to programming education using source code accumulated in OJ systems and related services. The proposed model is inspired by a neural language model with a pointer network (Bhoopchand et al., 2016; Li et al., 2017), and we demonstrate that the model outperforms a neural language model alone (White et al., 2015; Dam et al., 2016) also in the field of programming education.

AOJ accumulates a large quantity of various data, such as statistical information about the problems and the submission history of individual users, and the data are used in a variety of research. Saito and Watanobe (2020) proposed a learning path recommendation system based on a learner's ability charts by means of an

RNN. Intisar and Watanobe (2018a) proposed a method for the classification of OJ programmers based on rule extraction from a self-organising feature map, cluster analysis to estimate the difficulty of programming problems (Intisar and Watanobe, 2018b), and classification of programming problems based on topic modelling (Intisar et al., 2019). Teshima and Watanobe (2018) presented bug detection methods for the feedback system of an OJ system. Yoshizawa and Watanobe (2018) proposed a logic error detection algorithm based on structure patterns, which are an index of similarity based on abstract syntax trees, and error degree, which is a measure of appropriateness for feedback. Matsumoto and Watanobe (2019) analysed two different approaches, a static code analysis approach (Yoshizawa and Watanobe, 2018) and a deep learning approach (Teshima and Watanobe, 2018), through accumulated source codes for solving a programming task in an OJ system. Ohashi and Watanobe (2019) presented a method by which to classify source code based on convolutional neural networks. For the automatic generation of fill-in-the-blank programming problems, a method characterised by two principal elements, selection of exemplary source code and selection of places to be blanked, has been proposed (Terada and Watanobe, 2019). Rahman et al. (2020) proposed a sequential language model that uses an attention-mechanism-based LSTM neural network to assess and classify source code based on the estimated error probability. Other research activities and challenges which take advantage of OJ system to organise a learning ecosystem have been proposed in Watanobe et al. (2020).

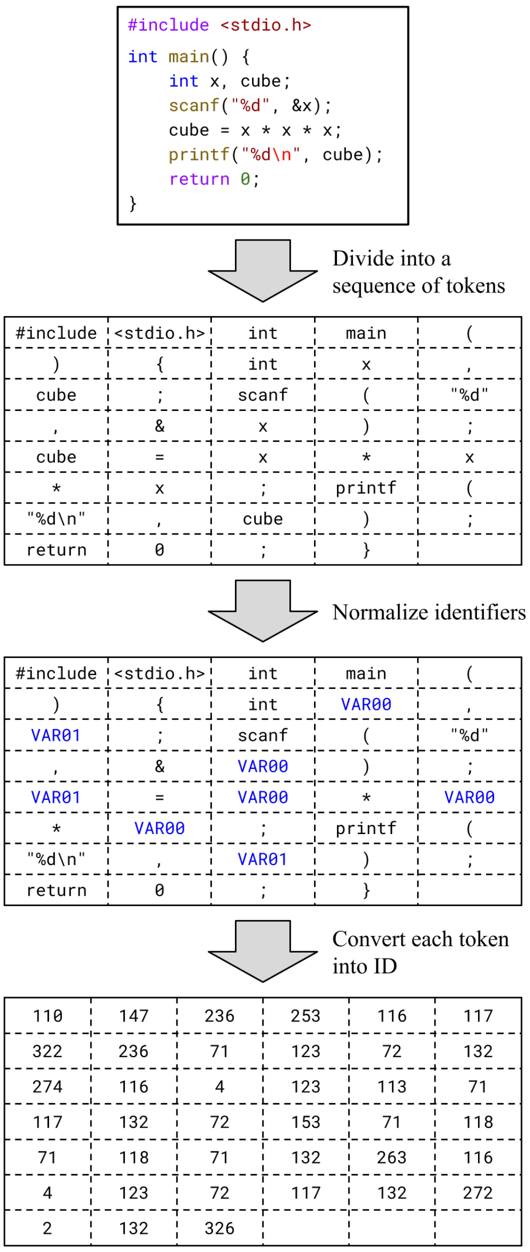
3 Data pre-processing

Figure 1 shows the flow of conversion from source code into a sequence so that the proposed models can receive input data in a unified format. First, in order to divide each of the collected source code lines into a sequence of words, we tokenise each source code using Tree-sitter, which is the universal parser of GitHub. When converting to the sequence, unnecessary information, including whitespace, blank lines, and comments, are all ignored. Whether to ignore line breaks and indents depends on the language characteristics of the source code being converted. For example, line breaks and indents are ignored if the code is written in C, but are not ignored in Python3, since line breaks indicate statement breaks and indentation determines the scope in Python3.

There are several types of identifiers in each programming language. For example, C has five types of identifiers: variable, function, argument, struct, and member. Similarly, Python3 also has five types of identifiers: variable, function, argument, class, and attribute. We normalise all identifiers in the sequence of words to express the types of identifiers explicitly. That is, we replace each identifier name with a generic identifier that indicates the corresponding identifier type concatenated with a number that makes the identifier unique within its scope, such as `$VAR06$`, `$FUNC02$`, and `$ARG04$`. Note that the numbers concatenated to each identifier type are in ascending order from 0 (e.g., `$VAR00$`, ..., `$VAR99$`), and only new identifiers defined within the file are replaced, i.e., identifiers that reference external APIs or libraries are not changed.

For converting each word in the sequence into the corresponding ID, we construct a vocabulary by concatenating a pre-determined word group and all of the collected words. A pre-determined word group consists primarily of `$PAD$` as padding, `$OOV$` as OoV, `$NUM$` as numerical constants, and `$STRING$` as strings. Depending on the language, the group also includes `$CHAR$` as a character, `$INDENT$` as an indent, and $OUTDENT$ as an outdent. We collect every word without duplication from every sequence of words in the dataset, excluding literals. After concatenating the above two word groups, each word is linked to a unique ID in ascending order from 0.

Figure 1 Flow of conversion from source code into sequence (see online version for colours)



```
#include <stdio.h>
int main() {
    int x, cube;
    scanf("%d", &x);
    cube = x * x * x;
    printf("%d\n", cube);
    return 0;
}
```

#include	<stdio.h>	int	main	(
)	{	int	x	,
cube	;	scanf	(	"%d"
,	&	x	)	;
cube	=	x	*	x
*	x	;	printf	(
"%d\n"	,	cube	)	;
return	0	;	}	

#include	<stdio.h>	int	main	(
)	{	int	VAR00	,
VAR01	;	scanf	(	"%d"
,	&	VAR00	)	;
VAR01	=	VAR00	*	VAR00
*	VAR00	;	printf	(
"%d\n"	,	VAR01	)	;
return	0	;	}	

110	147	236	253	116	117
322	236	71	123	72	132
274	116	4	123	113	71
117	132	72	153	71	118
71	118	71	132	263	116
4	123	72	117	132	272
2	132	326			

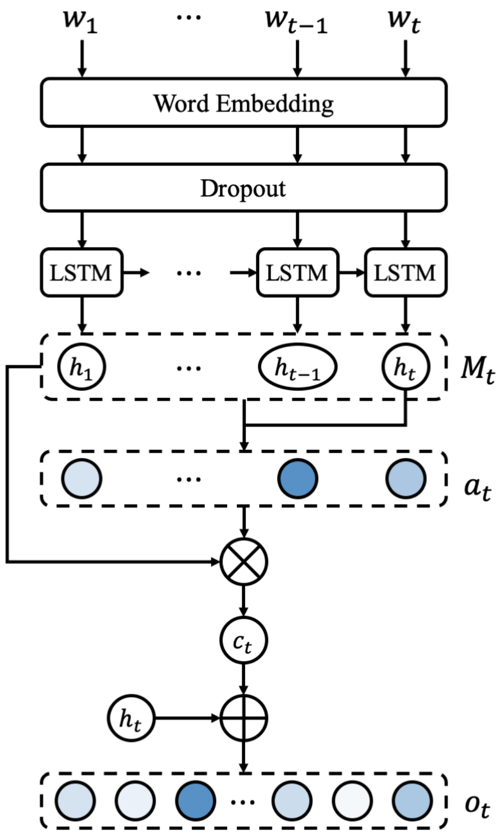
Finally, each word for every sequence is converted into the corresponding ID according to the constructed vocabulary. When we train the proposed models with one sequence of IDs $w_1, ..., w_n$, one sample consists of $w_1, ..., w_i$ as the input and w_{i+1} as the target, where $i \in \{1, ..., n-1\}$. In other words, the proposed models are trained in a chain reaction for one sequence of IDs.

4 Model architecture

4.1 Neural language model

Figure 2 shows the architecture of the model based on an LSTM network with an attention mechanism focusing on the prediction of the next within-vocabulary word. The sequence of IDs converted from an incomplete program constructed according to the process described in Section 3 is denoted by $w_1, ..., w_t$, and the model predicts w_{t+1}. The sequence, as the input, first flows to the word embedding layer, in which each word is mapped to a vector of real numbers. Due to data sparsity when representing words with unique and discrete IDs, vector representations via word embeddings are used, which helps partly to overcome this obstacle.

Figure 2 Architecture of neural language model (see online version for colours)



The vector representation flows to the dropout layer, which randomly zeroes some of the elements of the input vector with probability p using samples from a Bernoulli distribution. This is an effective technique for the regularisation and prevention of the co-adaptation of neurons (Hinton et al., 2012).

The dropped-out vector representation flows to the layer of an LSTM network, which is a special kind of RNN. The RNN is a type of neural network that operates on sequential data. The disadvantage of generic RNNs is that, because of vanishing gradients, RNNs fail to derive context from time steps that are substantially prior. In contrast, an LSTM network is able to remember long-term dependencies by enhancing the repeating module. Several gates that control the contribution of each memory unit help the network to remember better. A standard LSTM cell is defined by

$$h_t = f(x_t, h_{t-1}) \tag{1}$$

where x_t is the current input vector, h_{t-1} is the previous hidden state, and h_t is the current hidden state, which will be used to compute the prediction at time step t.

Next, an attention mechanism is applied to the hidden states obtained at all time steps in order to learn which words in the input are related to the output. Formally, by computing the relation between the last hidden state h_t and the external memory of previous hidden states, which is denoted by $M_t = [h_1, ..., h_t] \in \mathbb{R}^{k \times t}$, where k is the unit size of the hidden state, we obtain an attention distribution $a_t \in \mathbb{R}^{1 \times t}$. Then, by computing the weighted sum of the external memory M_t with the attention distribution a_t, we obtain a context vector $c_t \in \mathbb{R}^k$ as follows:

$$e_t = v^T \tanh(W_M M_t + (W_h h_t) 1_t^T) \tag{2}$$

$$a_t = \text{Softmax}(e_t) \tag{3}$$

$$c_t = M_t a_t^T \tag{4}$$

where $v \in \mathbb{R}^k$ and $W_M, W_h \in \mathbb{R}^{k \times k}$ are trainable parameters, and 1_t represents a t-dimensional vector of ones. The softmax function rescales an n-dimensional input vector so that the elements of the n-dimensional output vector lie in the range [0, 1] and sum to 1 and is defined as follows:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \quad (i = 1, ..., n) \tag{5}$$

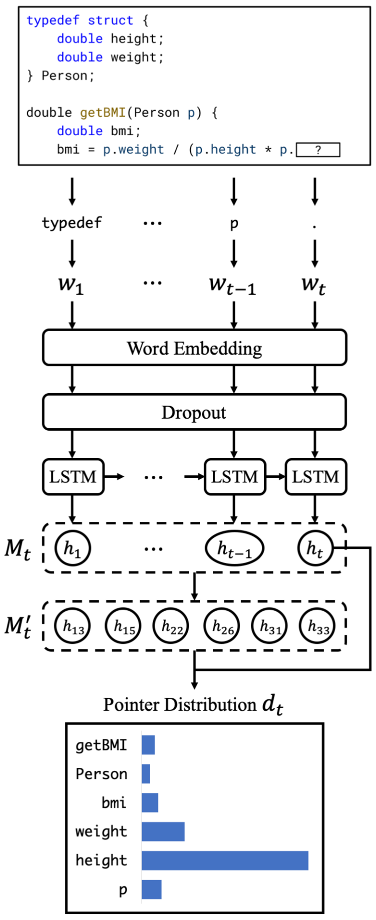
In order to obtain a distribution for the next word, we obtain the information about the next word based on the last hidden state h_t concatenated with the context vector c_t. The output vector $\tilde{h}_t \in \mathbb{R}^k$ is projected into the vocabulary space, and we apply a softmax function in order to obtain the final probability distribution $o_t \in \mathbb{R}^V$ for the next word w_{t+1} as follows:

$$\tilde{h}_t = \tanh\left(W_c \begin{bmatrix} h_t \\ c_t \end{bmatrix}\right) \tag{6}$$

$$o_t = \text{Softmax}(W_V \tilde{h}_t + b_V) \tag{7}$$

where $W_c \in \mathbb{R}^{k \times 2k}$ and $W_V \in \mathbb{R}^{V \times k}$ are trainable projection matrices, $b_V \in \mathbb{R}^V$ is a trainable bias vector, and V represents the size of the vocabulary.

Figure 3 Architecture of pointer model (see online version for colours)



```
typedef struct {
    double height;
    double weight;
} Person;

double getBMI(Person p) {
    double bmi;
    bmi = p.weight / (p.height * p. ?
```

4.2 *Pointer model*

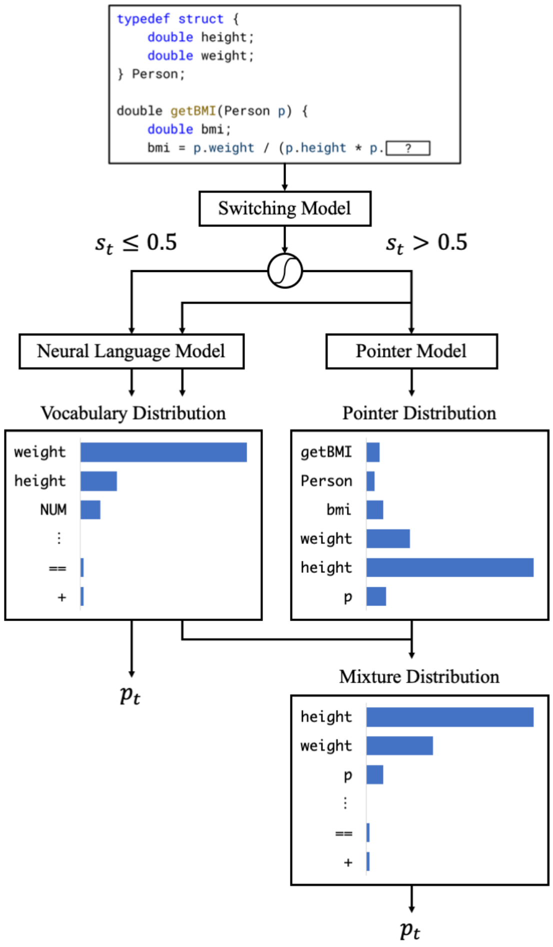
Figure 3 shows the architecture of the model based on a pointer network to a given incomplete program focusing on the prediction of the next referenceable identifier. The sequence of IDs constructed in Section 3 as the input flows in the order of the word embedding layer, the dropout layer, and the LSTM network layer, similar to the neural language model described in Subsection 4.1. Then, instead of the memory M_t consisting of all hidden states, the memory M'_t consisting only of hidden states where the corresponding word input at the time step is an identifier, is used as the attention window. If there are multiple identical identifiers in the input sequence, only the hidden state corresponding to the last appearing identifier is kept in the memory M'_t. Similar to the attention mechanism described in Subsection 4.1, we obtain a pointer distribution $d_t \in \mathbb{R}^L$, where L is the length of the memory M'_t by computing the relation between the last hidden state h_t and the memory M'_t, as follows:

$$e_t = v^T \tanh(W_M M'_t + (W_h h_t) 1_L^T) \tag{8}$$

$$d_t = \text{Softmax}(e_t) \tag{9}$$

The i^{th} term of the distribution d_t indicates the probability that the next identifier is the input word corresponding to the hidden state of $M'_t[i]$.

Figure 4 Architecture of pointer mixture model (see online version for colours)



```
typedef struct {
    double height;
    double weight;
} Person;

double getBMI(Person p) {
    double bmi;
    bmi = p.weight / (p.height * p.[ ? ]
```

4.3 Switching model

The switching model determines whether the next word is a referenceable identifier, i.e., this model takes the role of switching between the neural language model and the pointer model. Similar to the neural language model described in Subsection 4.1, the sequence of IDs constructed in Section 3 as the input flows in the order of the word embedding layer, the dropout layer, and the LSTM network layer. Then, $\tilde{h}_t \in \mathbb{R}^k$ obtained according to equations (2), (3), (4), and (6) is projected into a scalar value, and we apply a sigmoid function to obtain the probability $s_t \in [0, 1]$ indicating the probability that the next word w_{t+1} is a referenceable identifier, as follows:

$$s_t = \sigma(W_s \tilde{h}_t + b_s) \tag{10}$$

where $W_s \in \mathbb{R}^{1\times k}$ and $b_s \in \mathbb{R}^1$ are trainable parameters.

As shown in Figure 4, if s_t is less than or equal to the threshold, i.e., the probability that the next word is a referenceable identifier is low, the next word is predicted using the neural language model alone; otherwise, the next word is predicted using both the neural language model and the pointer model. In the former case, the vocabulary distribution o_t of the neural language model obtained in Subsection 4.1 is used as is for the prediction of the next word. In the latter case, in addition to the vocabulary distribution o_t of the neural language model, the pointer distribution d_t obtained through the pointer model in Subsection 4.2 is also taken into account in the prediction of the next word. In order to calculate the weighted sum of each distribution by s_t, the pointer distribution d_t is transformed into the vocabulary space as $o'_t \in \mathbb{R}^V$. Note that all identifier names are normalised in practice, such as `height`, `weight`, and `bmi` in Figure 4, are normalised to `MEMBER00`, `MEMBER01`, and `VAR00`, respectively. The final distribution $p_t \in \mathbb{R}^V$ for the next word is calculated as follows:

$$p_t = \begin{cases} o_t & \text{if } s_t \leq 0.5 \\ (1 - s_t)o_t + s_t o'_t & \text{if } s_t > 0.5 \end{cases} \tag{11}$$

where s_t denotes the probability to use the output of the pointer model and $1 - s_t$ denotes the probability to use the output of the neural language model.

5 Evaluation

5.1 Dataset

AOJ supports 18 different programming languages, including C, C++, Java, Ruby, and Python. We evaluate the proposed approach on two datasets: C as a statically typed language and Python3 as a dynamically typed language. These languages are widely used for beginners to get started with programming, and so the number of accumulated programs for each of these two languages is higher than that of other languages.

Table 1 Problem information for ITP

Problem ID	*Topic*	*Summary*
ITP1_1_A	Getting	Print ‘Hello World’ to standard output.
ITP1_1_B	started	Calculate the cube of a given integer x.
ITP1_1_C		Calculate the area and perimeter of a given rectangle.
ITP1_1_D		Convert a given integer S [seconds] to h:m:s.
ITP1_2_A	Branch on	Print small/large/equal relation of two given integers a and b.
ITP1_2_B	condition	Given three integers, print ‘yes’ if $a < b < c$, otherwise ‘no’.
ITP1_2_C		Print three given integers in ascending order.
ITP1_2_D		Determine whether the circle is arranged inside the rectangle.

Table 1 Problem information for ITP (continued)

Problem ID	*Topic*	*Summary*
ITP1_3_A	Repetitive processing	Print 'Hello World' to standard output 1,000 times.
ITP1_3_B		Print a given integer x for each dataset in the specified format.
ITP1_3_C		For each dataset, print two given integers in ascending order.
ITP1_3_D		Given three integers, print the number of divisors of c in $[a, b]$.
ITP1_4_A	Computation	Calculate and print a/b and its remainder as different data types.
ITP1_4_B		Calculate the area and circumference of a circle for given radius r.
ITP1_4_C		Given two integers a, b and an operator *op*, print the value of a *op* b.
ITP1_4_D		Print the min/max/sum for a given sequence of n integers.
ITP1_5_A	Structured program I	Draw an H [cm] $\times$ W [cm] rectangle with a single '#'.
ITP1_5_B		Draw an H [cm] $\times$ W [cm] frame with a single '#'.
ITP1_5_C		Draw an H [cm] $\times$ W [cm] chessboard by placing '#' and '.' alternately.
ITP1_5_D		Write a program equivalent to the specified program w/o *go to* statements.
ITP1_6_A	Array	Print a given sequence in the reverse order.
ITP1_6_B		Print the missing cards out of the given n playing cards.
ITP1_6_C		Given tenant/leaver notices, report the number of tenants for each room for 4 buildings, each with 3 floors, each with 10 rooms.
ITP1_6_D		Print the product of an $n \times m$ matrix A and an $m \times 1$ vector b.
ITP1_7_A	Structured program II	Given a list of student scores, evaluate the grade for each student.
ITP1_7_B		Print the number of combinations of three distinct integers that sum to x.
ITP1_7_C		Given an $r \times c$ table, print it and include a sum for each row/column.
ITP1_7_D		Print the product of an $n \times m$ matrix A and an $m \times l$ matrix B.
ITP1_8_A	Character	Convert upper/lower case letters to lower/upper case for a given string.
ITP1_8_B		Print the sum of digits of a given integer.
ITP1_8_C		Count the number of each alphabetical letter, ignoring the case.
ITP1_8_D		Find a pattern p in a ring-shaped text s.
ITP1_9_A	String	Print the number of times a word W appears in the text T.
ITP1_9_B		Shuffle a deck of n cards (a string) and print the final state.
ITP1_9_C		Read two cards n times, and report the final score of the game.
ITP1_9_D		Perform commands, including *reverse* and *replace*, to a given string.
ITP1_10_A	Math functions	Calculate the distance between two points: $P1(x_1, y_1), P2(x_2, y_2)$.
ITP1_10_B		For the given two sides of a triangle and the angle between them, calculate the area, circumference, and height of the triangle.
ITP1_10_C		Calculate the standard deviation of the scores $s_1, s_2, ...s_n$.
ITP1_10_D		Calculate the Minkowski's distance, where $p = 1, 2, 3$ and ∞.
ITP1_11_A	Structure and class	Construct a dice from given integers and simulate rolling the dice.
ITP1_11_B		Print the number on the right side of the rolled dice.
ITP1_11_C		Determine whether two given dice are identical.
ITP1_11_D		Determine whether all of the given n dice are different.

Table 2 Accuracies of next word prediction for the C language for each problem in ITP

Problem ID	*Programs*	*Tokens*	*Accuracy*			
			Neural language model		*Pointer mixture model*	
			w/o attention	*w/ attention*	*w/o attention*	*w/ attention*
ITP1_1_A	7,646	120,689	95.98%	*95.99*%	-	-
ITP1_1_B	6,292	226,485	95.68%	95.66%	*95.72*%	95.71%
ITP1_1_C	5,637	284,953	94.84%	94.78%	*94.90*%	94.81%
ITP1_1_D	4,452	287,492	92.01%	91.98%	*92.14*%	92.08%
ITP1_2_A	4,510	290,332	95.98%	95.98%	*95.99*%	*95.99*%
ITP1_2_B	4,309	258,454	*96.49*%	96.43%	96.47%	96.43%
ITP1_2_C	3,728	503,755	93.46%	93.48%	93.85%	*93.86*%
ITP1_2_D	3,247	312,330	93.11%	93.03%	*93.17*%	93.11%
ITP1_3_A	3,661	124,184	*94.91*%	94.81%	94.87%	94.78%
ITP1_3_B	3,106	191,049	91.15%	91.32%	91.36%	*91.43*%
ITP1_3_C	2,994	277,141	92.05%	92.01%	92.43%	*92.45*%
ITP1_3_D	2,860	221,352	93.19%	93.46%	93.41%	*93.51*%
ITP1_4_A	2,983	188,169	94.51%	94.61%	94.63%	*94.70*%
ITP1_4_B	2,405	117,917	92.31%	92.25%	*92.40*%	92.34%
ITP1_4_C	2,173	271,423	93.95%	94.18%	94.42%	*94.43*%
ITP1_4_D	2,286	290,739	88.81%	89.33%	89.94%	*90.00*%
ITP1_5_A	2,445	254,663	94.04%	94.34%	94.33%	*94.53*%
ITP1_5_B	2,321	342,523	92.51%	92.82%	93.09%	*93.19*%
ITP1_5_C	2,249	338,190	91.77%	92.48%	92.38%	*92.81*%
ITP1_5_D	1,136	134,685	89.37%	89.23%	*89.84*%	89.57%
ITP1_6_A	2,158	229,710	92.21%	92.43%	92.69%	*92.83*%
ITP1_6_B	1,812	497,960	90.88%	92.94%	92.61%	*93.15*%
ITP1_6_C	1,780	421,611	92.55%	93.36%	93.92%	*94.05*%
ITP1_6_D	1,515	305,383	93.12%	94.12%	94.24%	*94.50*%
ITP1_7_A	1,785	348,264	91.16%	92.82%	92.09%	*92.92*%
ITP1_7_B	1,773	261,458	89.94%	90.30%	90.87%	*90.94*%
ITP1_7_C	1,535	391,939	90.89%	92.87%	92.38%	*93.11*%
ITP1_7_D	1,309	386,090	93.41%	94.20%	94.51%	*94.79*%
ITP1_8_A	1,036	111,878	87.18%	86.98%	*87.92*%	87.75%
ITP1_8_B	1,071	109,338	87.09%	88.03%	88.44%	*88.71*%
ITP1_8_C	909	125,232	85.91%	86.89%	87.76%	*87.91*%
ITP1_8_D	761	107,209	84.14%	84.57%	86.15%	*86.17*%
ITP1_9_A	731	120,317	86.08%	87.52%	87.65%	*88.06*%
ITP1_9_B	616	117,161	82.39%	83.79%	85.50%	*85.81*%
ITP1_9_C	636	97,581	84.12%	85.06%	86.83%	*86.89*%
ITP1_9_D	495	141,168	78.43%	81.09%	84.50%	*85.57*%
ITP1_10_A	1,333	108,838	90.93%	90.93%	*91.33*%	91.23%
ITP1_10_B	716	89,370	86.00%	87.52%	88.17%	*88.58*%
ITP1_10_C	1,093	180,592	85.61%	88.24%	88.66%	*89.05*%
ITP1_10_D	1,008	294,048	86.46%	89.62%	89.06%	*90.06*%
ITP1_11_A	449	156,300	84.22%	88.63%	88.80%	*90.15*%
ITP1_11_B	345	171,160	78.11%	84.42%	82.76%	*85.90*%
ITP1_11_C	275	149,206	77.86%	82.02%	82.12%	*84.18*%
ITP1_11_D	209	127,074	71.01%	75.50%	77.33%	*80.25*%

Table 3 Accuracies of next word prediction for the Python3 language for each problem in ITP

Problem ID	*Programs*	*Tokens*	*Accuracy*			
			Neural language model		*Pointer mixture model*	
			w/o attention	*w/ attention*	*w/o attention*	*w/ attention*
ITP1_1_A	3,729	19,774	*99.61*%	99.58%	-	-
ITP1_1_B	2,916	51,926	94.60%	94.49%	*94.61*%	94.45%
ITP1_1_C	2,530	108,856	92.85%	92.91%	93.02%	*93.07*%
ITP1_1_D	2,147	110,542	90.23%	*90.30*%	*90.30*%	90.25%
ITP1_2_A	1,982	119,256	94.63%	*94.85*%	94.64%	94.84%
ITP1_2_B	1,899	98,394	95.35%	95.35%	95.35%	*95.37*%
ITP1_2_C	1,739	116,956	89.01%	88.84%	*89.56*%	89.49%
ITP1_2_D	1,565	123,671	90.81%	91.06%	91.11%	*91.34*%
ITP1_3_A	1,844	32,611	96.98%	*97.10*%	96.99%	*97.10*%
ITP1_3_B	1,562	83,275	91.76%	91.71%	*91.92*%	91.88%
ITP1_3_C	1,468	113,564	90.87%	90.90%	91.22%	*91.27*%
ITP1_3_D	1,458	96,299	93.23%	93.35%	*93.70*%	93.66%
ITP1_4_A	1,306	65,392	90.80%	90.54%	*90.94*%	90.80%
ITP1_4_B	1,304	52,191	90.01%	89.95%	*90.14*%	90.12%
ITP1_4_C	1,273	142,240	92.31%	92.59%	92.95%	*93.13*%
ITP1_4_D	1,351	84,797	90.11%	90.64%	90.86%	*91.16*%
ITP1_5_A	1,259	97,187	91.21%	91.22%	*91.52*%	91.49%
ITP1_5_B	1,000	113,685	89.07%	89.04%	*89.84*%	89.82%
ITP1_5_C	1,091	147,355	87.68%	88.42%	88.79%	*89.20*%
ITP1_5_D	709	64,114	88.11%	89.16%	88.72%	*89.52*%
ITP1_6_A	1,133	60,692	89.13%	*89.53*%	89.42%	89.52%
ITP1_6_B	916	162,798	82.50%	84.84%	84.74%	*85.79*%
ITP1_6_C	824	169,392	86.53%	88.36%	88.67%	*89.37*%
ITP1_6_D	834	122,695	88.36%	90.99%	91.49%	*92.18*%
ITP1_7_A	865	146,197	91.99%	93.08%	93.01%	*93.38*%
ITP1_7_B	758	93,019	86.58%	86.77%	87.52%	*87.72*%
ITP1_7_C	735	131,379	82.07%	83.53%	85.74%	*86.23*%
ITP1_7_D	645	132,934	85.57%	89.36%	90.25%	*91.79*%
ITP1_8_A	750	28,222	85.90%	85.52%	*86.36*%	86.02%
ITP1_8_B	772	41,897	88.08%	88.37%	88.87%	*88.90*%
ITP1_8_C	559	59,524	82.80%	84.06%	85.00%	*85.57*%
ITP1_8_D	684	38,052	84.90%	*85.66*%	84.86%	85.35%
ITP1_9_A	655	49,605	85.92%	86.51%	86.94%	*87.26*%
ITP1_9_B	626	56,549	85.40%	85.66%	86.64%	*86.79*%
ITP1_9_C	595	63,318	87.53%	88.51%	88.55%	*89.13*%
ITP1_9_D	530	102,318	84.00%	86.50%	86.07%	*87.51*%
ITP1_10_A	637	37,122	*87.64*%	87.01%	87.59%	87.08%
ITP1_10_B	551	63,014	85.03%	87.29%	86.91%	*88.36*%
ITP1_10_C	523	52,939	84.59%	85.74%	86.74%	*87.05*%
ITP1_10_D	490	89,204	79.20%	83.36%	83.20%	*85.27*%
ITP1_11_A	455	141,287	82.28%	86.11%	85.90%	*87.80*%
ITP1_11_B	321	146,048	75.24%	78.99%	78.09%	*80.90*%
ITP1_11_C	245	118,269	74.18%	77.67%	77.23%	*80.08*%
ITP1_11_D	148	76,336	70.36%	78.24%	73.24%	*79.28*%

For both C and Python3, all of the accepted source codes for each problem in introduction to programming (ITP), which is a problem set for novice programmers, are collected from the AOJ database. Table 1 shows the problem information for ITP, each with ID, topic, and summary. Source codes with more than 1,000 program words are excluded from the dataset.

5.2 Experimental setup

The three models described in Section 4 were developed in PyTorch (Paszke et al., 2019), with hyperparameters set by experience. The size of the vector representation for the word embedding is 100, the dropout probability for an element is 0.2 for the neural language model and the pointer model, and 0.5 for the switching model. The unit size of the hidden state for the LSTM network is 200. In order to train the models, a mini-batch stochastic gradient descent (SGD) is applied using the Adam optimisation algorithm with a learning rate of 0.001 (Kingma and Ba, 2014). In order to validate the training process for the neural language model and the pointer model, the cross-entropy loss (CEL) function is applied to each model as a loss function, which is defined by

$$E = -\sum_{k=1}^{K} y_k \log p_k \tag{12}$$

where K is the number of classes, y is a binary indicator (0 or 1) denoting whether the class label k is the correct classification for the observation, and p is the predicted probability that the observation is of class k. In order to validate the training process for the switching model, the binary cross-entropy loss (BCEL) function is applied to the model as a loss function, which is defined by

$$E = -y \log p - (1 - y) \log(1 - p) \tag{13}$$

where y is a binary indicator (0 or 1) denoting whether the next word is a referenceable identifier, and p is the predicted probability that the next word is a referenceable identifier. For each problem in AOJ, we collected programs that are solutions for the problem and shuffled them, then used 50% of them for training, 20% for validation, and 30% for testing. For example, assuming 1,000 programs are collected in the problem ITP1_1_A, we shuffle them and use 500 programs for training, 200 programs for validation, and 300 programs for testing. The models are prepared, trained and evaluated independently for each problem. For example, in the case of the problem ITP1_2_A, the untrained models go through the training process using the training and validation data for ITP1_2_A and are evaluated using the test data for ITP1_2_A. Their trained models are never used in any other problems than ITP1_2_A. The batch size is 32 with one program as one data entry, and each model is trained for 100 epochs. The parameters at the epoch with the lowest loss score for validation data are used for the evaluation.

5.3 Experimental results and discussion

In order to verify the accuracy improvement of next word prediction by incorporating an attention mechanism and a pointer network into a conventional neural language model, we examine the following four models:

- Neural language model alone *without* the attention mechanism.
- Neural language model alone *with* the attention mechanism.
- Pointer mixture model, which consists of the neural language model *without* the attention mechanism, the pointer model, and the switching model.
- Pointer mixture model, which consists of the neural language model *with* the attention mechanism, the pointer model, and the switching model.

Tables 2 and 3 show the accuracies of next word prediction for C and Python3 source codes, respectively, for each problem in ITP. Each row shows the problem ID, the total number of collected programs, the total number of program tokens, and the accuracy of each of the four models on the set of programs for testing. Note that italic text indicates the highest accuracy within the line, and accuracies for the pointer mixture model in ITP1_1_A are omitted because identifiers are not used for the problem.

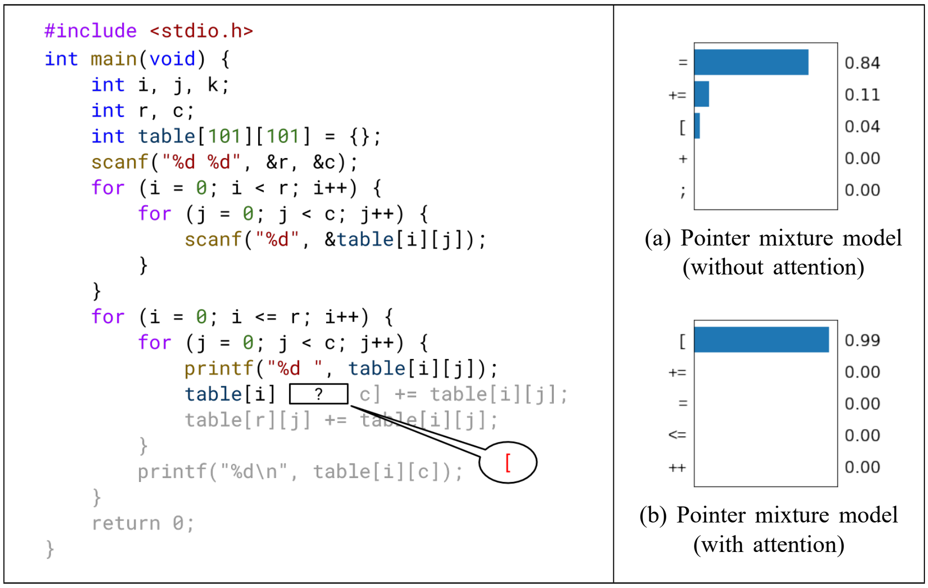
As shown in Table 2, the results focusing on the C language indicate that, for most problems, the pointer mixture model can predict the next words with higher accuracy than the neural language model alone, i.e., incorporating a pointer network makes it possible to predict the next referenceable identifiers more appropriately. Regarding the degree of accuracy improvement with respect to each problem topic, incorporating a pointer network increased the accuracy by 0.07% on average in ITP1_1, whereas the accuracy was increased by 3.72% on average in ITP1_11. This suggests that while a program for solving a problem becomes more complicated as the difficulty of the problem increases, a pointer network to the attention window composed only of the identifier information can address this complexity and assist in predicting the next referenceable identifiers. In approximately 75% of the problems, the accuracy of the next word prediction is improved by incorporating an attention mechanism. Since the improvement in accuracy has not been confirmed to a great extent, especially for topics ITP1_1 and ITP1_2, the higher the difficulty of problem topics, the more an attention mechanism tends to improve accuracy. The same is true within a problem topic because the difficulty level within that topic basically increases in order from problem A to problem D. For example, in topics ITP1_8 and ITP1_10, the improvement in accuracy has not been confirmed only in problem A. In comparison with the degree of accuracy improvement for each problem topic, in ITP1_11, which is the most difficult problem topic, incorporating an attention mechanism and a pointer network into the neural language model alone without an attention mechanism increased accuracy by 7.32% on average, which has brought the greatest improvement in all topics. Based on this observation, the pointer mixture model is very likely to improve the accuracy of word prediction, not only for problems for novice programmers, such as ITP, but also for problems for intermediate programmers that require knowledge of algorithms and data structures.

As shown in Table 3, the results focusing on Python3 indicate that the accuracy improvement of next word prediction has been confirmed in approximately 85% of the problems by incorporating a pointer network into the neural language model and in approximately 75% of the problems by incorporating an attention mechanism. In ITP1_11, the pointer mixture model with an attention mechanism has an average 6.5% increase in accuracy compared to the neural language model alone without an attention mechanism, which has brought about the greatest improvement in all topics. In addition, the pointer mixture model with an attention mechanism tends to have the highest accuracy among the four models as the problem topic progresses. From this, not only in statically typed languages, such as C, but also in dynamically typed languages, such as Python3, the more difficult a problem, i.e., the more complicated the program to solve the problem, the more an attention mechanism and a pointer network can contribute to improving the accuracy of next word prediction.

Tables 2 and 3 show that the pointer mixture model can serve as a code completion for all kinds and difficulties of problems with a higher accuracy than the conventional neural language model. However, when the code completion system is introduced into an e-learning system, there is a risk that learners are too dependent on the system to get the learning effect. As a premise, the model is supposed to be used in situations that could result in significantly reducing learning efficiency, such as when a learner wastes time being struggle and has no ideas. Actuality, user interfaces with the feedback functionalities are responsible for limiting and controlling the disclosure of the code completion. In addition to the interface, an embedded feedback system can compensate for the risk. Although the detail of the user interface and feedback system is out of scope of this paper, we present some suggestions as examples below:

- To provide some constraints to prevent learners from relying too much on a code completion system. For example, only when a learner does not type for a certain period of time, the system should show obtained suggestions.
- To provide a feedback system to enhance a learning effect. For example, after a submitted program is accepted by the judge, the feedback system can show the program with blank lines which are originally completed by the system and give a chance to fill in all the blanks as a review.

Figure 5 Example of code completion for the C language program for problem ITP1_7_C in AOJ (see online version for colours)



```
#include <stdio.h>
int main(void) {
    int i, j, k;
    int r, c;
    int table[101][101] = {};
    scanf("%d %d", &r, &c);
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &table[i][j]);
        }
    }
    for (i = 0; i <= r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", table[i][j]);
            table[i] [ ? ] c] += table[i][j];
            table[r][j] += table[i][j];
        }
        printf("%d\n", table[i][c]);
    }
    return 0;
}
```

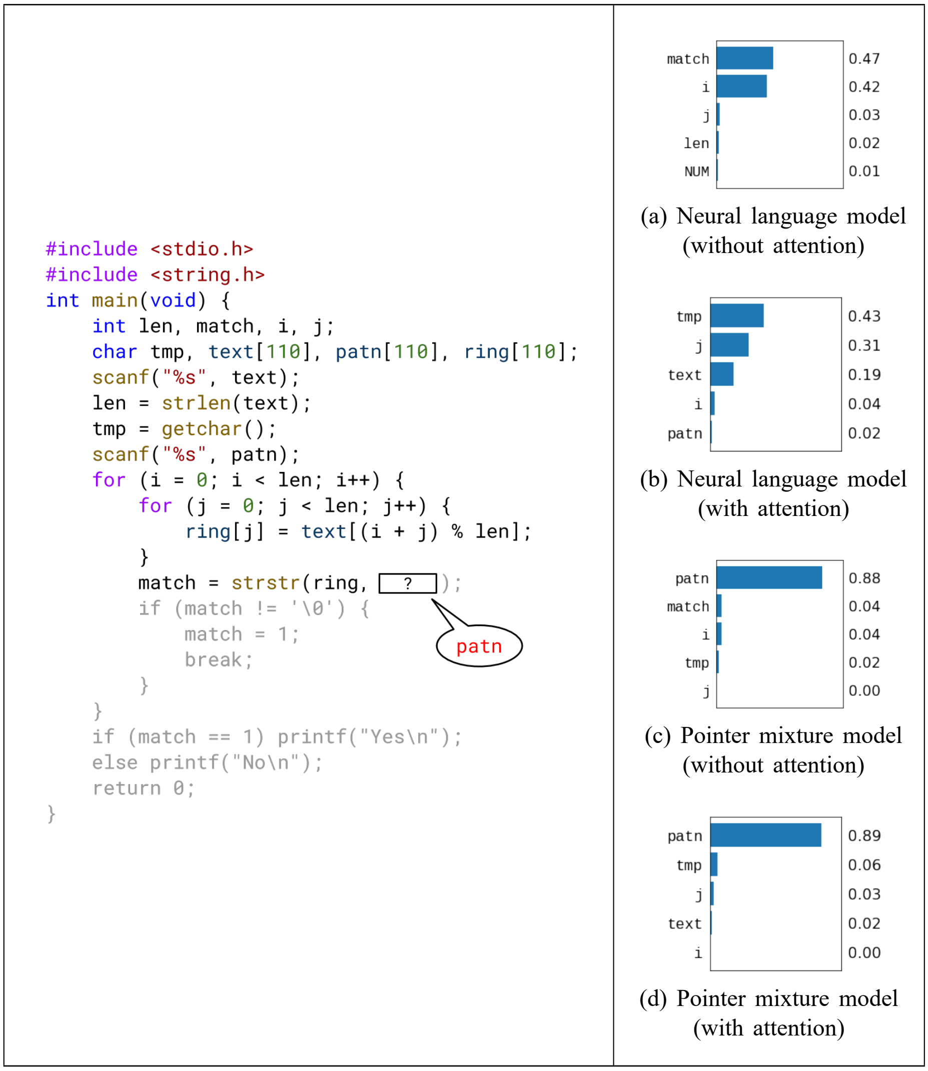
(a) Pointer mixture model (without attention)

(b) Pointer mixture model (with attention)

5.4 *Case study*

Figures 5 and 6 show code completion examples of C language programs for the problems ITP1_7_C and ITP1_8_D in AOJ, respectively. On the left side of each figure, an incomplete program for solving the problem is shown, and the models are required to correctly predict the next word (‘`[`’ in Figure 5 and ‘`patn`’ in Figure 6). The translucent code following the blank indicating the next word is only an example of a solution after the models have correctly predicted the next word. In other words, only the code before the blank exists in practice. On the right side of each figure, the results of the proposed models predicting the next word are shown as probability distributions, each of which shows the top five predictions.

Figure 6 Example of code completion for the C language program for problem ITP1_8_D in AOJ (see online version for colours)



```c
#include <stdio.h>
#include <string.h>
int main(void) {
    int len, match, i, j;
    char tmp, text[110], patn[110], ring[110];
    scanf("%s", text);
    len = strlen(text);
    tmp = getchar();
    scanf("%s", patn);
    for (i = 0; i < len; i++) {
        for (j = 0; j < len; j++) {
            ring[j] = text[(i + j) % len];
        }
        match = strstr(ring,  ?  );
        if (match != '\0') {
            match = 1;
            break;
        }
    }
    if (match == 1) printf("Yes\n");
    else printf("No\n");
    return 0;
}
```

(a) Neural language model (without attention)

(b) Neural language model (with attention)

(c) Pointer mixture model (without attention)

(d) Pointer mixture model (with attention)

In Figure 5, the task of the problem is to read the number of rows r, columns c, and a table of $r \times c$ integer elements and to print a new table, which includes the total sum for each row and column. The switching model has correctly predicted that the next word is not a referenceable identifier, i.e., the output probability has been $s_t \leq 0.5$, so the pointer mixture model has applied the neural language model alone for next word prediction. The model without an attention mechanism incorrectly predicts '`=`' as the next word, whereas that with an attention mechanism can correctly predict '`[`' as the next word with high probability, i.e., can understand that the variable `table` is a two-dimensional array and some processing will be performed on one element in the i^{th} row of the array variable. As such, previous information can be effectively used to predict the next word by incorporating an attention mechanism.

In Figure 6, the task of the problem is to read a ring-shaped text s and a pattern p, and print 'yes' in a line if p is in s, otherwise 'no'. Regardless of whether an attention mechanism is incorporated, the neural language model alone fails to predict the next word, whereas the pointer mixture model can correctly predict the next word '`patn`' with high probability. This indicates that the switching model correctly predicts that the next word is a referenceable identifier, and the pointer mixture model can predict the next word with an emphasis on the pointer model.

6 Conclusions

Herein, we have presented a methodology for code completion that has two key constituents: the prediction of the next within-vocabulary word and the prediction of the next referenceable identifier. We have proposed a model for the former based on an LSTM network with an attention mechanism, a model for the latter based on a pointer network to a given incomplete program, and a model for switching between the former and latter models. The proposed algorithms have been demonstrated using data from AOJ. As a result, in the field of not only software engineering but also programming education, the pointer mixture model succeeded in predicting both the next within-vocabulary word and the referenceable identifier with higher accuracy than the conventional neural language model alone in both statically and dynamically typed languages. The present research can be expected to contribute to support for programming learning for beginners. In addition, the proposed prediction method is very likely to be applicable to source code in a variety of other languages, regardless of whether the language is statically or dynamically typed, such as Java and Ruby. Generally, the algorithm can be applied to code completion tasks not only in AOJ but also in other OJ systems, where accumulated source codes are available.

References

Bahdanau, D., Cho, K. and Bengio, Y. (2015) 'Neural machine translation by jointly learning to align and translate', in *3rd International Conference on Learning Representations*.

Bhoopchand, A., Rocktäschel, T., Barr, E.T. and Riedel, S. (2016) 'Learning Python code suggestion with a sparse pointer network', *CoRR*, abs/1611.08307.

Chung, M. and Kim, J. (2016) 'The internet information and technology research directions based on the fourth industrial revolution', *KSII Transactions on Internet and Information Systems*, Vol. 10, No. 3, pp.1311–1320.

Dam, H.K., Tran, T. and Pham, T. (2016) 'A deep language model for software code', *CoRR*, abs/1608.02715.

Ginzberg, A., Kostas, L. and Sengendo, J.M. (2017) *Automatic Code Completion*, Technical Report, Stanford CS224n Class Project.

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2012) 'Improving neural networks by preventing co-adaptation of feature detectors', *CoRR*, abs/1207.0580.

Hochreiter, S. and Schmidhuber, J. (1997) 'Long short-term memory', *Neural Computation*, Vol. 9, No. 8, pp.1735–1780.

Intisar, C.M. and Watanobe, Y. (2018a) 'Classification of online judge programmers based on rule extraction from self organizing feature map', in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, pp.313–318.

Intisar, C.M. and Watanobe, Y. (2018b) 'Cluster analysis to estimate the difficulty of programming problems', in *Proceedings of the 3rd International Conference on Applications in Information Technology*, pp.23–28.

Intisar, C.M., Watanobe, Y., Poudel, M. and Bhalla, S. (2019) 'Classification of programming problems based on topic modeling', in *Proceedings of the 2019 7th International Conference on Information and Education Technology*, pp.275–283.

Jia, Y. (2019) 'Attention mechanism in machine translation', *Journal of Physics: Conference Series*, Vol. 1314, p.12186.

Kingma, D. and Ba, J. (2014) 'Adam: a method for stochastic optimization', *International Conference on Learning Representations*.

Li, J., Wang, Y., King, I. and Lyu, M.R. (2017) 'Code completion with neural attention and pointer networks', *CoRR*, abs/1711.09573.

Luong, M., Pham, H. and Manning, C.D. (2015) 'Effective approaches to attention-based neural machine translation', *CoRR*, abs/1508.04025.

Matsumoto, T. and Watanobe, Y. (2019) 'Towards hybrid intelligence for logic error detection', in *The 18th International Conference on Intelligent Software Methodologies, Tools, and Techniques (SoMeT)*, pp.120–131.

Mei, H., Bansal, M. and Walter, M.R. (2016) 'Coherent dialogue with attention-based language models', *CoRR*, abs/1611.06997.

Ohashi, H. and Watanobe, Y. (2019) 'Convolutional neural network for classification of source codes', in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp.194–200.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. (2019) 'PyTorch: an imperative style, high-performance deep learning library', in *Advances in Neural Information Processing Systems*, Vol. 32, pp.8026–8037, Curran Associates, Inc., Vancouver, Canada.

Rahman, M.M., Watanobe, Y. and Nakamura, K. (2020) 'Source code assessment and classification based on estimated error probability using attentive LSTM language model and its application in programming education', *Applied Sciences*, Vol. 10, No. 8, p.2973.

Saito, T. and Watanobe, Y. (2020) 'Learning path recommendation system for programming education based on neural networks', *International Journal of Distance Education Technologies*, Vol. 18, No. 1, pp.36–64.

Terada, K. and Watanobe, Y. (2019) 'Automatic generation of fill-in-the-blank programming problems', in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp.187–193.

Teshima, Y. and Watanobe, Y. (2018) 'Bug detection based on LSTM networks and solution codes', in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp.3541–3546.

Vinyals, O., Fortunato, M. and Jaitly, N. (2015) ‘Pointer networks’, in *Advances in Neural Information Processing Systems*, Vol. 28, pp.2692–2700, Curran Associates, Inc., Montreal, Canada.

Wang, L. and Liu, P. (2019) ‘A sentiment analysis model based on attention mechanism and compound model’, in *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pp.203–206.

Wasik, S., Antczak, M., Badura, J., Laskowski, A. and Sternal, T. (2018) ‘A survey on online judge systems and their applications’, *ACM Comput. Surv.*, Vol. 51, No. 1, pp.1–34.

Watanobe, Y., Chowdhury, I., Cortez, R. and Vazhenin, A. (2020) ‘Next-generation programming learning platform: architecture and challenges’, *SHS Web of Conferences*, Vol. 77, p.1004.

Watanobe, Y. (2015) ‘Development and operation of an online judge system’, *IPSJ Magazine*, Vol. 56, No. 10, pp.998–1005.

Watanobe, Y. (2018) *Aizu Online Judge* [online] https://onlinejudge.u-aizu.ac.jp/ (accessed February 2020).

White, M., Vendome, C., Linares-Vásquez, M. and Poshyvanyk, D. (2015) ‘Toward deep learning software repositories’, in *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp.334–345.

Yoshizawa, Y. and Watanobe, Y. (2018) ‘Logic error detection algorithm for Novice programmers based on structure pattern and error degree’, in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, pp.297–301.

Zhong, C., Yang, M. and Sun, J. (2019) ‘JavaScript code suggestion based on deep learning’, in *Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence*, pp.145–149.