Improving Automatic Source Code Summarization via Deep Reinforcement Learning

Yao Wan, Zhou Zhao
College of Computer Science and Technology, Zhejiang University, Hangzhou, China
{wanyao,zhaozhou}@zju.edu.cn

Min Yang
Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
min.yang@siat.ac.cn

Guandong Xu
Advanced Analytics Institute, University of Technology Sydney, Sydney, Australia
guandong.xu@uts.edu.au

Haochao Ying, Jian Wu
College of Computer Science and Technology, Zhejiang University, Hangzhou, China
wujian2000@zju.edu.cn
haochaoying@zju.edu.cn

Philip S. Yu
University of Illinois at Chicago, Illinois, USA
Institute for Data Science, Tsinghua University, Beijing, China
psyu@uic.edu

ABSTRACT

Code summarization provides a high level natural language description of the function performed by code, as it can benefit the software maintenance, code categorization and retrieval. To the best of our knowledge, most state-of-the-art approaches follow an encoder-decoder framework which encodes the code into a hidden space and then decode it into natural language space, suffering from two major drawbacks: a) Their encoders only consider the sequential content of code, ignoring the tree structure which is also critical for the task of code summarization; b) Their decoders are typically trained to predict the next word by maximizing the likelihood of next ground-truth word with previous ground-truth word given. However, it is expected to generate the entire sequence from scratch at test time. This discrepancy can cause an *exposure bias* issue, making the learnt decoder suboptimal. In this paper, we incorporate an abstract syntax tree structure as well as sequential content of code snippets into a deep reinforcement learning framework (i.e., actor-critic network). The actor network provides the confidence of predicting the next word according to current state. On the other hand, the critic network evaluates the reward value of all possible extensions of the current state and can provide global guidance for explorations. We employ an advantage reward composed of BLEU metric to train both networks. Comprehensive experiments on a real-world dataset show the effectiveness of our proposed model when compared with some state-of-the-art methods.

CCS CONCEPTS

• **Software and its engineering** → **Documentation**; • **Computing methodologies** → **Natural language generation**;

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238206

KEYWORDS

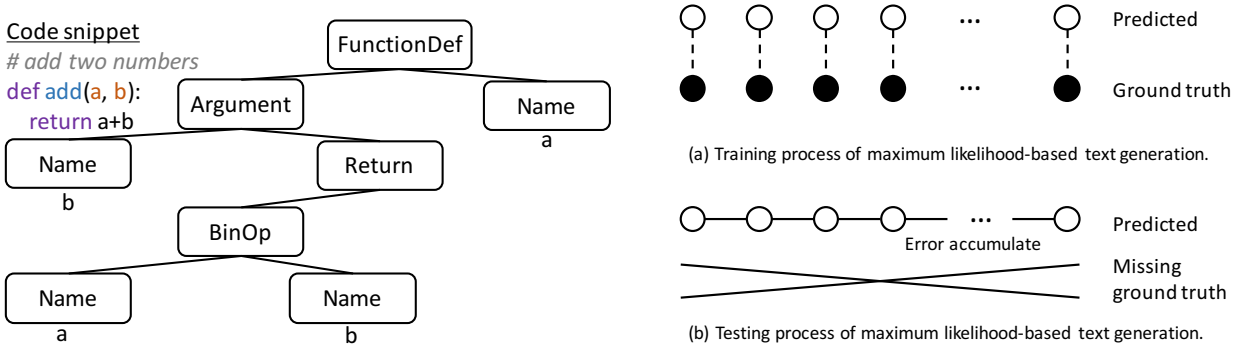
Code summarization, comment generation, deep learning, reinforcement learning

ACM Reference Format:
Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238206

1 INTRODUCTION

In the life cycle of software development (e.g., implementation, testing and maintenance), nearly 90% of effort is used for maintenance, and much of this effort is spent on understanding the maintenance task and related software source codes [22]. Thus, documentation which provides a high level description of the task performed by code is always a must for software maintenance. Even though various techniques have been developed to facilitate the programmer during the implementation and testing of software, documenting code with comments remains a labour-intensive task, making few real-world software projects adequately document the code to reduce future maintenance costs [9, 16]. It's nontrivial for a novice programmer to write good comments for source codes. A good comment should at least has the following characteristics: a) Correctness. The comments should correctly clarify the intent of code. b) Fluency. The comments should be fluent natural languages that can be easily read and understood by maintainers. c) Consistency. The comments should follow a standard style/format for better code reading. Code summarization is a task that tries to comprehend code and automatically generate descriptions directly from the source code. The summarization of code can also be viewed as a form of document expansion. Successful code summarization can not only benefit the maintenance of source codes [15, 30], but also be used to improve the performance of code search using natural language queries [32, 51] and code categorization [31].

Motivation. Recent research has made inroads towards automatic generation of natural language descriptions of software. As far as



(a) An example of abstractive syntax tree (AST). **(b) The limitation of maximum likelihood based text generation.**

Figure 1: An illustration of the motivation of our paper. Traditional methods suffer from the following two limitations: a) On representing the code, the structure information of code is always ignored. b) Traditional maximum likelihood based methods suffer from the exposure bias issue.

we know, most of existing code summarization methods learn the semantic representation of source codes based on statistical language models [30, 33], and then generate comments based on templates or rules [42]. With the development of deep learning, some neural translation models [2, 13, 15] have also been introduced for code summarization, which mainly follow an encoder-decoder framework. They generally employ recurrent neural networks (RNN e.g., LSTM [14]) to encode the code snippets and utilize another RNN to decode that hidden state to coherent sentences. These models are typically trained to maximize the likelihood of the next word on the assumption that previous words and ground-truth are given. These models are limited from two aspects: a) The code sequential and structural information is not fully utilized on feature representation, which is critical for code understanding. For example, given two simple expressions "`f=a+b`" and "`f=c+d`", although they are quite different as two lexical sequences, they share the same structure (e.g., abstractive syntax tree). b) These models, also termed "teacher-forcing", suffer from the *exposure bias* since in testing time the ground-truth is missing and previously generated words from the trained model distribution are used to predict the next word [37]. Figure 1(b) presents a simple illustration of the discrepancy among training and testing process in these classical encoder-decoder models. In the testing phase, this exposure bias makes error accumulated and makes these models suboptimal, not able to generate those words which are appropriate but with low probability to be drawn in the training phase.

Contribution. In this paper, we aim to address these two mentioned issues. To effectively capture the structural (or syntactic) information of code snippets, we employ abstract syntax tree (AST) [7], a data structure widely used in compilers, to represent the structure of program code. Figure 1a shows an example of Python code snippet and its corresponding AST. The root node is a composite node of type `FunctionDef`, while the leaf nodes which are typed as `Name` are tokens of code snippets. It's worth mentioning that the tokens from AST parsing may be different from those from word segmentation. In our paper, we consider both of them. We parse the code snippets into ASTs, and then propose an AST-based LSTM model [46] to represent the structure of code. We also use another LSTM model [14] to represent the sequential information of code. Besides, we apply a hybrid attention layer to fuse the structure representation and sequential representation of code on predicting the word, considering the alignment between predicted word and source word.

To overcome the exposure bias, we draw on the insights of deep reinforcement learning, which integrates exploration and exploitation into a whole framework. Instead of learning a sequential recurrent model to greedily look for the next correct word, we utilize an actor network and a critic network to jointly determine the next best word at each time step. The actor network, which provides the confidence of predicting the next word according to current state, serves as a local guidance. The critic network, which evaluates the reward value of all possible extensions of the current state, serves as a global guidance. Our framework is able to include the good words that are with low probability to be drawn by using the actor network alone. To learn these two networks more efficiently, we start with pretraining an actor network using standard supervised learning with cross entropy loss, and pretraining a critic network with mean square loss. Then, we update the actor and critic networks according to the advantage reward composed of BLEU metric via policy gradient. We summarize our main contributions as follows.

- We propose a more comprehensive representation method for source code, with one AST-based LSTM for the structure of source code, and another LSTM for the sequential content of source code. Furthermore, a hybrid attention layer is applied to fuse these two representations.
- To the best of our knowledge, it is the first time that we propose an advanced deep reinforcement learning framework, named actor-critic network, to cope with the exposure bias issue existing in most traditional maximum likelihood-based code summarization frameworks.
- We validate our proposed model on a real-world dataset of 108,726 Python code snippets. Comprehensive experiments show the effectiveness of the proposed model when compared with some state-of-the-art methods.

Organization. The remainder of this paper is organized as follows. We provide some background knowledge on neural language model, RNN encoder-decoder model and reinforcement learning in Section 2 for a better understanding of our proposed model. We also formally define the problem in Section 2. Section 3 gives an overview of our proposed framework. Section 4 presents a hybrid embedding approach for code representation. Section 5 shows our proposed deep reinforcement learning framework (i.e., actor-critic network). Section 6 describes the dataset used in our experiment and shows the experimental results and analysis. Section 7 shows some threats to validity and limitations existing in our model. Section 8 highlights some works related to this paper. Finally, we conclude this paper in Section 9.

2 BACKGROUND

As we declared before, the code summarization task can be seen as a text generation task given the source code. In this section, we first present some background knowledge on text generation which will be used in this paper, including language model, attentional RNN encoder-decoder model and reinforcement learning for better decoding. To start with, we introduce some basic notations and terminologies. Let $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$ denote a sequence of source code snippet, $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$ denote a sequence of generated words, where $|\cdot|$ denotes the length of sequence. Let T denote the maximum step of decoding in the encoder-decoder framework. We will often use notation $\mathbf{y}_{m\ldots l}$ to refer to subsequences of the form $(y_m, \ldots, y_l)$. $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$ is the training dataset, where N is the size of training set.

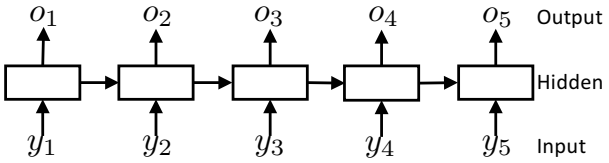
2.1 Language Model

Language model computes the probability of occurrence of a number of words in a particular sequence. The probability of a sequence of T words $\{y_1, \ldots, y_T\}$ is denoted as $p(y_1, \ldots, y_T)$. Since the number of words coming before a word, y_i , varies depending on its location in the input document, $p(y_1, \ldots, y_T)$ is usually conditioned on a window of n previous words rather than all previous words:

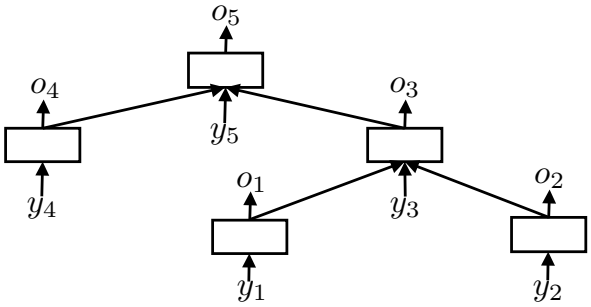
$$p(y_{1:T}) = \prod_{i=1}^{i=T} p(y_i|y_{1:i-1}) \approx \prod_{i=1}^{i=T} p(y_i|y_{i-(n-1):i-1}). \quad (1)$$

This kind of n-grams approach suffers apparent limitations [27, 39]. For example, the n-gram model probabilities can not be derived directly from the frequency counts, because models derived this way have severe problems when confronted with some n-grams that have not been explicitly seen before.

The neural language model is a language model based on neural networks. Unlike the n-gram model which predicts a word based on a fixed number of predecessor words, a neural language model can predict a word by predecessor words with longer distances. Figure 2(a) shows the basic structure of a RNN. The neural network includes three layers, that is, an input layer which maps each word to a vector, a recurrent hidden layer which recurrently computes and updates a hidden state after reading each word, and an output layer which estimates the probabilities of the following word given the current hidden state. The RNN reads the words in the sentence one by one, and predicts the possible following word at each time step. At step t, it estimates the probability of the following word



(a) The structure of recurrent neural network (RNN)



(b) The structure of tree-stuctured RNN (Tree-RNN)

Figure 2: RNN and Tree-RNN (adapted from [46]).

$p(y_{t+1}|y_{1:t})$ by the following steps: First, the current word y_t is mapped to a vector by the input layer e. Then, it generates the hidden state $\mathbf{h}_t$ at time t according to the previous hidden state $\mathbf{h}_{t-1}$ and the current input y_t:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, e(y_t)). \quad (2)$$

Here, two common options for f are long short-term memory (LSTM) [14] and the gated recurrent unit (GRU) [21]. Finally, the $p(y_{t+1}|y_{1:t})$ is predicted according to the current hidden state $\mathbf{h}_t$:

$$p(y_{t+1}|y_{1:t}) = g(\mathbf{h}_t), \quad (3)$$

where g is a stochastic output layer (typically a softmax for discrete outputs) that generates output tokens.

2.2 Attentional RNN Encoder-Decoder Model

RNN encoder-decoder has two recurrent neural networks. The encoder transforms the code snippet $\mathbf{x}$ into a sequence of hidden states $(\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{|\mathbf{x}|})$ with a RNN, while the decoder uses another RNN to generate one word y_{t+1} at a time in the target space.

2.2.1 Encoder. As a RNN, the encoder has a hidden state, which is a fixed-length vector. At the time step t, the encoder computes the hidden state $\mathbf{h}_t$ by:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{c}_{t-1}, e(\mathbf{x}_t))). \quad (4)$$

Here, f is the hidden layer which has two main options, i.e., LSTM and GRU. The last symbol of $\mathbf{x}$ should be an end-of-sequence ($< eos >$) symbol which notifies the encoder to stop and output the final hidden state $\mathbf{h}_T$, which is used as a vector representation of $\mathbf{x}$.

2.2.2 Decoder. The output of the decoder is the target sequence $\mathbf{y} = (y_1, \cdots, y_T)$. One input of the decoder is a $< start >$ symbol denoting the beginning of the target sequence. At the time step t, the decoder computes the hidden state $\mathbf{h}_t$ and the conditional distribution of the next symbol y_{t+1} by:

$$p(y_{t+1}|y_t) = g(\mathbf{h}_t, \mathbf{c}_t), \quad (5)$$

where g is a stochastic output layer and $\mathbf{c}_t$ is the distinct context vector for y_t, computed by:

$$\mathbf{c}_t = \sum_{j=1}^{|\mathbf{x}|} \alpha_{t,j} \mathbf{h}_j, \tag{6}$$

where $\alpha_{t,j}$ is the attention weight of y_t on $\mathbf{h}_j$ [4].

2.2.3 Training Goal. The encoder and decoder networks are jointly trained to maximize the following objective:

$$\max_{\theta} \mathcal{L}(\theta) = \max_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \log p(\mathbf{y}|\mathbf{x};\theta), \tag{7}$$

where θ is the set of the model parameters. We can see that this classical encoder-decoder framework targets on maximizing the likelihood of ground-truth word conditioned on previously generated words. As we have mentioned above, the maximum likelihood based encoder-decoder framework suffers the exposure bias issue. Motivated by this, we introduce the reinforcement learning technique for better decoding.

2.3 Reinforcement Learning for Better Decoding

The reinforcement learning is an approach that interacts with the real environment and learns the optimal policy from the reward signal. It tries to generate text from scratch without ground truth in the testing phase. Under this approach, the text generation process can be viewed as a Markov Decision Process (MDP) $\{S, A, P, R, \gamma\}$. In the MDP setting, state $\mathbf{s}_t$ at time step t consists of the source code snippets $\mathbf{x}$ and the words/actions predicted until t, $y_0, y_1, \ldots, y_t$. The action space is the dictionary $\mathcal{Y}$ that the words are drawn from, *i.e.*, $y_t \subset \mathcal{Y}$. With the definition of the state, the state transition function P is $\mathbf{s}_{t+1} = \{\mathbf{s}_t, y_{t+1}\}$, where the action y_{t+1} becomes a part of the next state $\mathbf{s}_{t+1}$ and the reward r_{t+1} is received. $\gamma \in [0, 1]$ is the discount factor. The objective of generation process is to find a policy that maximizes the expected reward of generation sentence sampled from the model's policy:

$$\max_{\theta} \mathcal{L}(\theta) = \max_{\theta} \mathbb{E}_{\substack{\mathbf{x}\sim\mathcal{D} \\ \hat{\mathbf{y}}\sim P_\theta(\cdot|\mathbf{x})}} [R(\hat{\mathbf{y}}, \mathbf{x})], \tag{8}$$

where θ is the parameter of policy needed to be learnt, $\mathcal{D}$ is the training set, $\hat{\mathbf{y}}$ is the predicted actions/words, and R is the reward function. Our problem can be formulated as follows.

> Given a code snippet $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$, our goal is to find a policy that generates a sequence of words $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$ from dictionary $\mathcal{Y}$ with the objective of maximizing the expected reward.

To learn the policy, many approaches have been proposed, which are mainly categorized into two classes [44]. a) The policy-based approaches (e.g., REINFORCE [50]) which optimizes the policy directly via policy gradient. b) The value-based approaches (e.g., Q-learning [48]) which learns the Q-function, and in each time the agent selects the action with highest Q-value. It has been verified that the policy-based methods may suffer from a variance issue and the value-based methods may suffer from a bias issue [17]. Thus in our paper, we adopt the actor-critic learning method which is a more advanced technique that has the advantage of both policy- and value-based methods.

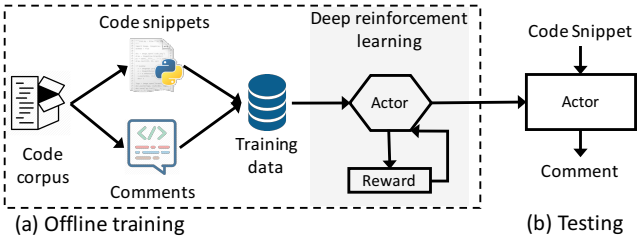


Figure 3: An overall workflow of getting a trained model.

3 OVERVIEW OF PROPOSED FRAMEWORK

In this section, we firstly have a simple overview on the workflow of how to get a trained model for code summarization. Then we present an overview of the network architecture of our proposed deep reinforcement learning based model. Figure 3 shows the overall workflow of how to get a trained model. It includes an offline training stage and an online summarization stage. In the training stage, we prepare a large-scale corpus of annotated $< code, comment >$ pairs. The annotated pairs are then fed into our proposed deep reinforcement learning model for training. After training, we can get a trained actor network. Then, given a code snippet, corresponding comment can be generated by the trained actor network.

Figure 4 is an overview of the network architecture of our proposed deep reinforcement learning based model. The architecture of our model follows the actor-critic framework [19], which has been successfully adopted in the decision-making scenarios such as AlphaGo [41]. We split the framework into four submodules. (a) Hybrid code representation (cf. Sec. 4). This module is used to represent the source code into a hidden space, which is also called encoder in the encoder-decoder framework. (b) Hybrid attention (cf. Sec. 5.1.1). On decoding the encoded hidden space into the comment space, the attention layer is used to assign different weights to the code snippet tokens for better generation. (c) Text generation (cf. Sec. 5.1.2). This module is a RNN-based generative network, which is used to generate the next word based on previous generated words. (d) Critic (cf. Sec. 5.2). This module is used to evaluate whether the generated word is good or not.

Since the generated tokens on (d) can also been seen as actions, we can also called the process (a)-(b)-(c) as actor network. Compared with the architecture of traditional encoder-decoder framework, our proposed model has an additional critic module used to evaluate the value of action taken under current state. The process (a)-(b)-(c)-(d) can also be called as critic network. We can see that the actor and critic networks share the modules (a)-(b)-(c), reducing the number of learning parameters a lot. We will elaborate each component in this framework in the following sections.

4 HYBRID REPRESENTATION OF CODE

Different from previous methods that just utilize sequential tokens to represent code, we also consider the structure information of source code. In this section, we present a hybrid embedding approach for code representation. We apply an LSTM to represent

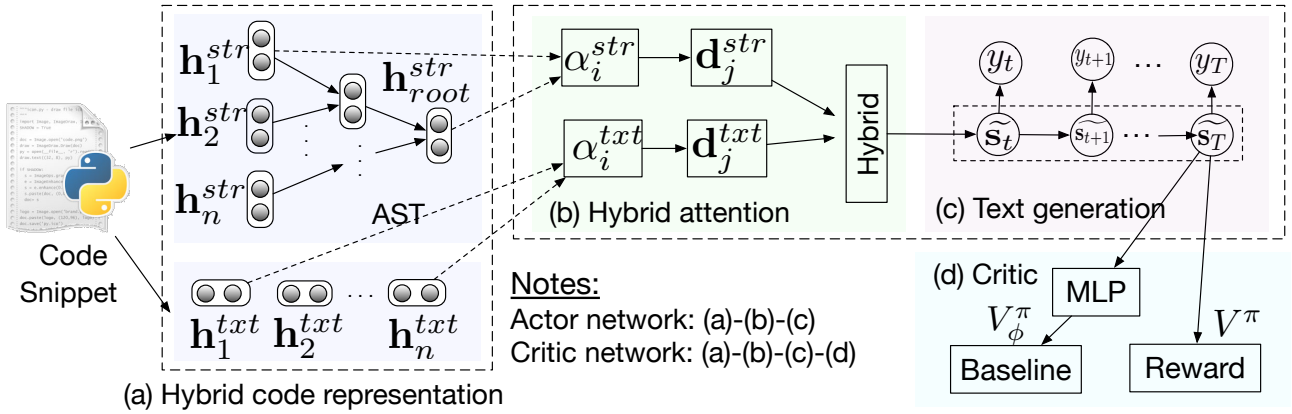


Figure 4: An overview of our proposed deep reinforcement learning framework for code summarization.

the lexical level of code, and an AST-based LSTM to represent the syntactic level of code.

4.1 Lexical Level

The key insight into lexical level representation of source code is that comments are always extracted from the lexical of code, such as the function name, variable name and so on. It's apparent that we apply an RNN (e.g., LSTM) to represent the sequential information of source code. In our paper, the LSTM is adopted.

4.2 Syntactic Level

In executing a program, a compiler decomposes a program into constituents and produces intermediate code according to the syntax of the language. AST is one type of intermediate code that represents the hierarchical syntactic structure of a program [1]. We represent the syntactic level of source code from the aspect of AST embedding. Similar to a traditional LSTM unit, we propose AST-based LSTM where the LSTM unit also contains an input gate, a memory cell and an output gate. Different from a standard LSTM unit which only has one forget gate for its previous unit, an AST-based LSTM unit contains multiple forget gates. Given an AST, for any node j, let the hidden state and memory cell of its l-th child be h_{jl} and c_{jl} respectively. Refer to [46], the hidden state is updated as follows:

$$
\begin{aligned}
\mathbf{i}_j &= \sigma(\mathbf{W}^{(i)}\mathbf{x}_j + \sum_{l=1}^{N}\mathbf{U}_l^{(i)}\mathbf{h}_{jl} + \mathbf{b}^{(i)}),\\
\mathbf{f}_{jk} &= \sigma(\mathbf{W}^{(f)}\mathbf{x}_j + \sum_{l=1}^{N}\mathbf{U}_{kl}^{(f)}\mathbf{h}_{jl} + \mathbf{b}^{(f)}),\\
\mathbf{o}_j &= \sigma(\mathbf{W}^{(o)}\mathbf{x}_j + \sum_{l=1}^{N}\mathbf{U}_l^{(o)}\mathbf{h}_{jl} + \mathbf{b}^{(o)}),\\
\mathbf{u}_j &= \tanh(\mathbf{W}^{(u)}\mathbf{x}_j + \sum_{l=1}^{N}\mathbf{U}_l^{(u)}\mathbf{h}_{jl} + \mathbf{b}^{(u)}),\\
\mathbf{c}_j &= \mathbf{i}_j \odot \mathbf{u}_j + \sum_{l=1}^{N}\mathbf{f}_{jl} \odot \mathbf{c}_{jl},\\
\mathbf{h}_j &= \mathbf{o}_j \odot \tanh(\mathbf{c}_j),
\end{aligned}
\tag{9}
$$

where $k = 1, 2, \cdots, N$. Each of $\mathbf{i}_j$, $\mathbf{f}_{jk}$, $\mathbf{o}_j$ and $\mathbf{u}_j$ denotes an input gate, a forget gate, an output gate, and a state for updating the memory cell, respectively. $\mathbf{W}^{(\cdot)}$ and $\mathbf{U}^{(\cdot)}$ are weight matrices, $\mathbf{b}^{(\cdot)}$ is a bias vector, and $\mathbf{x}_j$ is the word embedding of the j-th node. $\sigma(\cdot)$ is the logistic function, and the operator $\odot$ denotes element-wise multiplication between vectors. It's worth mentioning that when the tree is simply a chain, namely $N = 1$, the AST-based LSTM unit reduces to the standard LSTM. Figure 2 shows the structure of RNN and Tree-RNN.

Notice that the number of children N varies for different nodes of different ASTs, which may cause problem in parameter-sharing. For simplification, we transform the generated ASTs to binary trees by the following two steps which have been adopted in [49]: a) Split nodes with more than 2 children, generate a new right child together with the old left child as its children, and then put all children except the leftmost as the children of this new node. Repeat this operation in a top-down way until only nodes with 0, 1, 2 children left; b) Combine nodes with 1 child with its child.

5 DEEP REINFORCEMENT LEARNING FOR CODE SUMMARIZATION

In this section, we introduce the advanced deep learning framework named actor-critic network, which has been successfully used in the AlphaGo [41]. We introduce the actor and critic network respectively and then present how to train them simultaneously.

5.1 Actor Network

After obtaining the representation of code snippet, we need to decode it into comment. Here we describe how we generate comment from the hidden space with a hybrid attention layer.

5.1.1 Hybrid Attention. Different parts of the code make different contributions to the final output of comment. We adopt an attention mechanism [4] which has been successfully used in neural machine translation. In the attention layer, we have two attention scores, one $\alpha_t^{str}(j)$ for structural representation and another $\alpha_t^{txt}(j)$ for sequential representation of code. At t-th step of the decoding process, the attention scores $\alpha_t^{str}(j)$ and $\alpha_t^{txt}(j)$ are calculated as follows:

$$\alpha_t^{str}(j) = \frac{\exp(\mathbf{h}_j^{str} \cdot \mathbf{s}_t)}{\sum_{k=1}^{n} \exp(\mathbf{h}_k^{str} \cdot \mathbf{s}_t)}, \quad \alpha_t^{txt}(j) = \frac{\exp(\mathbf{h}_j^{txt} \cdot \mathbf{s}_t)}{\sum_{k=1}^{n} \exp(\mathbf{h}_k^{txt} \cdot \mathbf{s}_t)}, \tag{10}$$

where n is the number of code tokens; $\mathbf{h}_j^{(\cdot)} \cdot \mathbf{s}_t$ is the inner project of $\mathbf{h}_j^{(\cdot)}$ and $\mathbf{s}_t$, which is used to directly calculate the similarity score between $\mathbf{h}_j^{(\cdot)}$ and $\mathbf{s}_t$. The t-th context vector $\mathbf{d}_t^{(\cdot)}$ is calculated as the summarization vector weighted by $\alpha_t^{(\cdot)}(j)$:

$$\mathbf{d}_t^{str} = \sum_{t=1}^{n} \alpha_t^{str}(j)\mathbf{h}_j^{str}, \quad \mathbf{d}_t^{txt} = \sum_{t=1}^{n} \alpha_t^{txt}(j)\mathbf{h}_j^{txt}. \tag{11}$$

To integrate the structural context vector and the textual vector, we concatenate them firstly and then feed them into an one-layer linear network:

$$\mathbf{d}_t = \mathbf{W}_{\mathbf{d}_t}[\mathbf{d}_t^{str}; \mathbf{d}_t^{txt}] + \mathbf{b}_{\mathbf{d}_t}), \tag{12}$$

where $[\mathbf{d}_t^{str}; \mathbf{d}_t^{txt}]$ is the concatenation of $\mathbf{d}_t^{str}$ and $\mathbf{d}_t^{txt}$. The context vector is then used for the $(t+1)$-th word prediction by putting an additional hidden layer $\widetilde{\mathbf{s}_t}$:

$$\widetilde{\mathbf{s}_t} = \tanh(\mathbf{W}_c[\mathbf{s}_t; \mathbf{d}_t] + \mathbf{b}_d), \tag{13}$$

where $[\mathbf{s}_t; \mathbf{d}_t]$ is the concatenation of $\mathbf{s}_t$ and $\mathbf{d}_t$.

5.1.2 Text Generation. The model predicts the t-th word by using a softmax function. Let p_π denote a policy π determined by the actor network, $p_\pi(y_t|\mathbf{s}_t)$ denote the probability distribution of generating t-th word y_t, we can get the following equation:

$$p_\pi(y_t|\mathbf{s}_t) = softmax(\mathbf{W}_s\widetilde{\mathbf{s}_t} + \mathbf{b}_s). \tag{14}$$

5.2 Critic Network

Unlike traditional encoder-decoder framework that generates sequence directly via maximizing likelihood of next word given the ground truth word, we directly optimize the evaluation metrics such as BLEU [34] for code summarization. We apply a critic network to approximate the value of generated action at time step t. Different from the actor network, this critic network outputs a single value instead of a probability distribution on each decoding step. Before introducing critic network, we introduce the value function.

Given the policy π, sampled actions and reward function, the value function V^π is defined as the prediction of total reward from the state $\mathbf{s}_t$ at step t under policy π, which is formulated as follows:

$$V^\pi(\mathbf{s}_t) = \mathbb{E}_{\substack{s_{t+1:T}, \\ y_{t:T}}}\left[\sum_{l=0}^{T-t} r_{t+l}|y_{t+1}, \cdots, y_T, \mathbf{h}\right], \tag{15}$$

where T is the max step of decoding; $\mathbf{h}$ is the representation of code snippet. For code summarization, we can only obtain an evaluation score (BLEU) when the sequence generation process (or episode) is finished. The episode terminates when step exceeds the max-step T or generating the end-of-sequence (EOS) token. Therefore, we define the reward as follows:

$$r_t = \begin{cases} 0 & t < T \\ BLEU & t = T \text{ or } EOS \end{cases}. \tag{16}$$

Mathematically, the critic network tries to minimize the following loss function, where mean square error is used.

$$\mathcal{L}(\phi) = \frac{1}{2}\left\|V^\pi(\mathbf{s}_t) - V_\phi^\pi(\mathbf{s}_t)\right\|^2, \tag{17}$$

where $V^\pi(\mathbf{s}_t)$ is the target value, $V_\phi^\pi(\mathbf{s}_t)$ is the value predicted by critic network and ϕ is the parameter of critic network.

5.3 Model Training

We use the policy gradient method to optimize policy directly, which is widely used in reinforcement learning. For actor network, the goal of training is to minimize the negative expected reward, which can be defined as $\mathcal{L}(\theta) = -\mathbb{E}_{y_1,\ldots,_T\sim\pi}(\sum_{l=t}^{T} r_t)$, where θ is the parameter of actor network. Denote all the parameters as $\Theta = \{\theta, \phi\}$, the total loss of our model can be represented as $\mathcal{L}(\Theta) = \mathcal{L}(\theta) + \mathcal{L}(\phi)$.

For policy gradient, it is typically better to train an expression of the following form according to [40]:

$$\nabla_\theta \mathcal{L}(\Theta) = \mathbb{E}[\sum_{t=0}^{T-1} A^\pi(\mathbf{s}_t, y_{t+1})\nabla_\theta \log \pi_\theta(y_{t+1}|\mathbf{s}_t)], \tag{18}$$

where $A^\pi(\mathbf{s}_t, y_{t+1})$ is advantage function. The reason why we choose advantage function is that it achieves smaller variance when compared with some other ones such as TD residual and reward with baseline [40].

According to the definition of advantage function, we can formulate the advantage function as follows. One can refer to [40] for more details.

$$A^\pi(\mathbf{s}_t, y_t) = Q^\pi(\mathbf{s}_t, y_t) - V^\pi(\mathbf{s}_t), \tag{19}$$

where $Q^\pi(\mathbf{s}_t, y_t)$ is the state-action value function which is defined as $Q^\pi(\mathbf{s}_t, y_t) = \mathbb{E}_{\substack{s_{t+1:T}, \\ y_{t+1:T}}}\left[\sum_{l=0}^{T-t} r_{t+l}\right]$. From this formulation, we can find that the advantage function measures whether or not the action is better or worse than the policy's default behavior. Therefore, a step in the policy gradient direction can increase the probability of better-than-average actions and decrease the probability of worse-than-average actions.

On the other hand, the gradient of critic network is calculated as follows:

$$\nabla_\phi \mathcal{L}(\Theta) = \sum_{t=0}^{T-1}[V^\pi(\mathbf{s}_t) - V_\phi^\pi(\mathbf{s}_t)]\nabla_\phi V_\phi^\pi(\mathbf{s}_t). \tag{20}$$

We employ stochastic gradient descend with the diagonal variant of AdaGrad [10] to optimize the parameters of our framework. Algorithm 1 summarizes our proposed model described above.

6 EXPERIMENTS AND ANALYSIS

To evaluate our proposed approach, in this section, we conduct experiments to answer the following questions:

- **RQ1.** Does our proposed approach improve the performance of code summarization when compared with some state-of-the-art approaches?
- **RQ2.** What's the effectiveness of each component for our proposed model? For example, what about the performance

Algorithm 1 Actor-Critic training for code summarization.

1: Initialize actor $\pi_{y_{t+1|s_t}}$ and critic $V^{\pi}(s_t)$ with random weights θ and ϕ;
2: Pre-train the actor to predict ground truth y_t given $\{y_1, \cdots, y_{t-1}\}$ by minimizing Eq. 7;
3: Pre-train the critic to estimate $V(s_t)$ with fixed actor;
4: **for** $t = 1 \rightarrow T$ **do**
5: Receive a random example, and generate sequence of actions $\{y_1, \cdots, y_T\}$ according to current policy π_θ;
6: Calculate advantage estimate A^{π} according to Eq. 19;
7: Update critic weights ϕ using the gradient in Eq. 20;
8: Update actor weights θ using the gradient in Eq. 18.

of hybrid code representation and reinforcement learning respectively?

- **RQ3.** What's the performance of our proposed model on the datasets with different code or comment length?

We ask RQ1 to evaluate our deep reinforcement learning-based model compared to some state-of-the-art baselines. We ask RQ2 in order to evaluate each component of our model. We ask RQ3 to evaluate our model when varying the length of code or comment. In the following subsections, we first describe the dataset, some evaluation metrics and the training details. Then, we introduce some baselines for RQ1. Finally, we report our results and analysis for the research questions.

6.1 Dataset Preparation

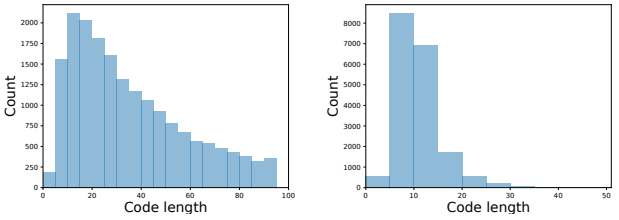
We evaluate the performance of our proposed method using the dataset in [6], which is obtained from a popular open source projects hosting platform, GitHub[1]. The dataset contains 108,726 code-comment pairs. The vocabulary size of code and comment is 50,400 and 31,350, respectively. For cross-validation, We shuffle the dataset and use the first 60% for training, 20% for validation and the remaining for testing. To construct the tree-structure of code, we parse Python code into abstract syntax trees via ast[2] lib. To convert code into sequential text, we tokenize the code by {. , " ' : ;) (! (space)}, which has been used in [31]. We tokenize the comment by {(space)}.

Figure 5 shows the length distribution of code and comment on testing data. From Figure 5a, we can find that the lengths of most code snippets are located between 20 to 60. This verifies the quote in [26] "Functions should hardly ever be 20 lines long". In Python language, the limited length should be shorter. From Figure 5b, we can notice that the length of nearly all the comments are between 5 to 15. This reveals the comment sequence that we need to generate will not be too long.

6.2 Evaluation Metrics

We evaluate the performance of our proposed model based on four widely-used evaluation criteria in the area of neural machine translation and image captioning, i.e., BLEU [34], METEOR [5], ROUGE-L [23] and CIDER [47]. BLEU measures the average n-gram precision on a set of reference sentences, with a penalty for short sentences. METEOR is recall-oriented and measures how well our model captures content from the references in our output. ROUGE-L takes into account sentence level structure similarity naturally and identifies longest co-occurring in sequence n-grams automatically. CIDER is a consensus based evaluation protocol for image captioning.

[1]https://github.com/
[2]https://docs.python.org/2/library/ast.html



(a) Code length distribution. **(b) Comment length distribution.**

Figure 5: Length distribution of testing data.

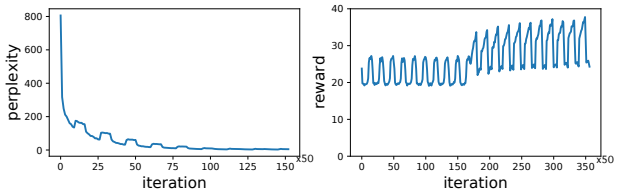


Figure 6: Iteration of training perplexity and reward.

6.3 Training Details

The hidden size of the encoder and decoder LSTM networks are both set to be 512, and the word embedding size is set to be 512. The mini-batch size is set to be 64, while the learning rate is set to be 0.001. We pretrain both actor network and critic network with 10 epochs each, and train the actor-critic network simultaneously 10 epoches. We record the perplexity[3]/reward every 50 iterations. Figure 6 shows the perplexity and reward curves of our method. All the experiments in this paper are implemented with Python 2.7, and run on a computer with an 2.2 GHz Intel Core i7 CPU, 64 GB 1600 MHz DDR3 RAM, and a Titan X GPU with 12 GB memory, running Ubuntu 16.04.

6.4 RQ1: Compared to Baselines

We compare our model with the following baselines:

- Seq2Seq [43] is a classical encoder-decoder framework in neural machine translation, which encodes the source sentences into a hidden space, and decodes it into target sentences. In our comparison, the encoder and decoder are both based on LSTM.
- Seq2Seq+Attn [4] is a derived version of Seq2Seq model with an attentional layer for word alignment.
- Tree2Seq [49] follows the same architecture with Seq2Seq and applies AST-based LSTM as encoder for the task of code clone detection.

[3]Perplexity is a function of cross entropy loss, which has been widely used in evaluation of many natural language processing tasks.

Table 1: Comparison of the overall performance between our model and previous methods. (Best scores are in boldface.)

	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	CIDER
Seq2Seq	0.1660	0.0251	0.0100	0.0056	0.0535	0.2838	0.1262
Seq2Seq+Attn	0.1897	0.0419	0.0200	0.0133	0.0649	0.3083	0.2594
Tree2Seq	0.1649	0.0236	0.0096	0.0053	0.0501	0.2794	0.1168
Tree2Seq+Attn	0.1887	0.0417	0.0197	0.0129	0.0644	0.3068	0.2331
Hybrid2Seq+Attn+DRL (Our)	**0.2527**	**0.1033**	**0.0640**	**0.0441**	**0.0929**	**0.3913**	**0.7501**

Table 2: Effectiveness of each component for our proposed model. (Best scores are in boldface.)

	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	CIDER
Seq2Seq+Attn+DRL	0.2421	0.0919	0.0513	0.0325	0.0882	**0.3935**	0.6390
Tree2Seq+Attn+DRL	0.2309	0.0854	0.0499	0.0338	0.0843	0.3767	0.6060
Hybrid2Seq	0.1837	0.0379	0.0183	0.0122	0.0604	0.3020	0.2223
Hybrid2Seq+Attn	0.1965	0.0516	0.0280	0.0189	0.0693	0.3154	0.3475
Hybrid2Seq+Attn+DRL (Our)	**0.2527**	**0.1033**	**0.0640**	**0.0441**	**0.0929**	0.3913	**0.7501**

- Tree2Seq+Attn [11] is a derived version of Tree2Seq model with an attentional layer, which has been applied in neural machine translation
- Hybrid2Seq(+Attn+DRL) represents three versions of our proposed model with/without Attn/DRL component.

Table 1 shows the experimental results of comparison between our proposed model and some previous ones. From this table, we can find that our proposed model outperforms other baselines in almost all of evaluation metrics. When comparing Seq2Seq/Tree2Seq with its correspond attention-based version, we can see that attention is really effective in aligning the code tokens with comment tokens. We can also find that the performance of simply encoding the tree structure of code is worse than that of simply encoding the code as sequence. This can be illustrated by that the words of comments are always drawn from the tokens of code. Thus, our model which considers both the structure and sequential information of code achieves the best performance in this comparison.

6.5 RQ2: Component Analysis

Table 2 shows the effectiveness of some main components in our proposed model. From this table, comparing the results of Seq2Seq+Attn/Tree2Seq+Attn with and without (Table 1) deep reinforcement learning (DRL), we can see that the proposed DRL component can really boost the performance of comment generation for source code. We can also find the proposed approach of integrating the LSTM for content and AST-based LSTM for structure is effective on representing the code as compared with the corresponding non-hybrid ones in Table 1. Furthermore, it also verifies that our proposed hybrid attention mechanism works well in our model.

6.6 RQ3: Parameter Analysis

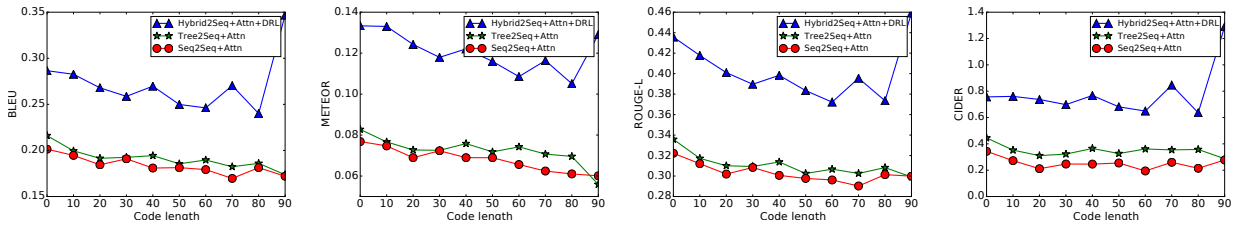
We vary the length of code and comment since the code length may have an effect on the representation of code and the comment length may have an effect on the performance of text generation. Figure 7 and Figure 8 show the performance of our proposed method when compared with two baselines on datasets of varying code lengths and comment lengths, respectively.

From Figure 7, we can see that our model performs best when compared with other baselines on four metrics with respect to different code lengths. Additionally, we can see that the our proposed model has a stable performance even though the code length increases dramatically. We attribute this effect to the hybrid representation we adopt in our model. For Figure 8, recall the comment length distribution in Figure 5b. Since nearly all the comment lengths of testing data are under 20, we ignore the performance analysis over the samples whose comment length are larger than 20. From this figure, we can see the performances of our model and baselines vary dramatically on four metrics with respect to different comment lengths.

6.7 Qualitative Analysis and Visualization

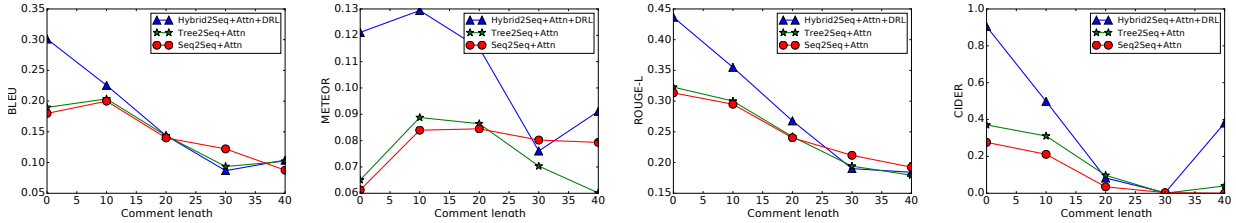
We show two examples in Table 3. It's clear that the generated comments by our model are closest to the ground truth. Although those models without DRL can generate some tokens which are also in the ground truth, they can't predict those tokens which are not frequently appeared in the training data. On the contrary, our deep reinforcement learning based model can generate some tokens which are closer to the ground truth, like `git`, `symbolic`. This can be illustrated by the fact that our model has a more comprehensive exploration on the word space and optimizes the BLEU score directly.

In Table 3, we also visualize two attentions in our proposed model for the target sentences. For example, for Case 1 with target sentence *check if git is installed* ., we can notice that the str-attn (left of figure) focuses more on tokens like `OSError`, `False`, `git`, `version`, which represent the structure of code. On the other hand, the attention of txt-attn (right of figure) is comparatively dispersed, and have a focus on some tokens like `def`, which is of little significance for code summarization. This verifies our assumption that LSTM can capture the sequential content of code, and AST-based LSTM can capture the structure information of code. Thus, it's reasonable to fuse them together for a better representation.



(a) BLEU (b) METEOR (c) ROUGE-L (d) CIDER

Figure 7: Experimental results of our proposed method and some baselines on different metrics w.r.t. varying code length.



(a) BLEU (b) METEOR (c) ROUGE-L (d) CIDER

Figure 8: Experimental results of our proposed method and some baselines on different metrics w.r.t. varying comment length.

7 THREATS TO VALIDITY AND LIMITATIONS

One threat to validity is that our approach is experimented only on Python code collected from GitHub, so they may not be representative of all the comments. However, Python is a popular programming language used in a large number of projects. In the future, we will extend our approach to other programming languages. Another threat to validity is on the metrics we choose for evaluation. It has always been a tough challenge to evaluate the similarity between two sentences for the tasks such as neural machine translation [43], image captioning [18]. In this paper, we only adopt four popular automatic metrics, it is necessary for us to evaluate the performance of generated text from more perspectives, such as human evaluation. Furthermore, in the deep reinforcement learning perspective, we only set the BLEU score of generated sentence as the reward. It's well known that for a reinforcement learning method, one of the biggest challenge is how to design a reward function to measure the value of action correctly, and it is still an open problem. In our future work, we plan to devise a reward function that can reflect the value of each action more correctly.

8 RELATED WORK

8.1 Deep Code Representation

With the successful development of deep learning, it has also become more and more prevalent for representing source code in the domain of software engineering research. Gu et al. [12] use a sequence-to-sequence deep neural network [43], originally introduced for statistical machine translation, to learn intermediate distributed vector representations of natural language queries which they use to predict relevant API sequences. Mou et al. [29] learn distributed vector representations using custom convolutional neural networks to represent features of snippets of code, then they assume that student solutions to various coursework problems have been intermixed and seek to recover the solution-to-problem mapping via classification. Li et al. [21] learn distributed vector representations for the nodes of a memory heap and use the learned representations to synthesize candidate formal specifications for the code that produces the heap. Piech et al. [36] and Parisotto et al. [35] learn distributed representations of source code input/output pairs and use them to assess and review student assignments or to guide program synthesis from examples. Neural code-generative models of code also use distributed representations to capture context, which is a common practice in natural language processing. For example, the work of Maddison and Tarlow [25] and other neural language models (e.g. LSTMs in Dam et al. [8]) describe context distributed representations while sequentially generating code. Ling et al. [24] and Allamanis et al. [3] combine the code-context distributed representation with distributed representations of other modalities (e.g. natural language) to synthesize code.

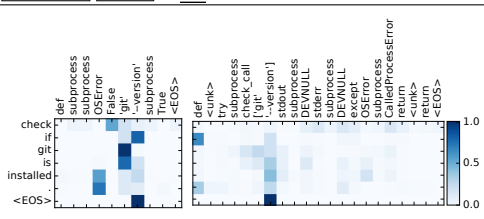
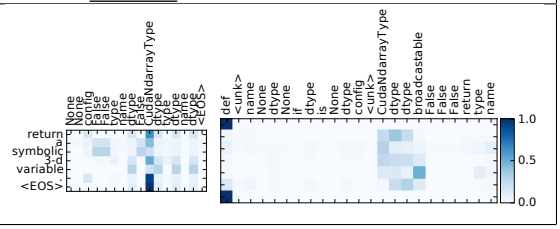
8.2 Source Code Summarization

Code summarization is a novel task in the area of software engineering and has drawn great attention in recent years. The existing works for code summarization can be mainly categorized as rule based approaches [42], statistical language model based approaches [30] and deep learning based approaches [2, 13, 15]. Sridhara et al. [42] construct a software word usage model first, and generate comment according to the tokenized function/variable names via rules. Movshovitz-Attias et al. [30] predict comments from Java source files using topic models and n-grams. In [2], the authors introduce an attentional neural network that employs convolution on the input tokens to detect local time-invariant and long-range topical attention features to summarize source code snippets into short, descriptive function name-like summaries. Iyer et al. [15] propose to use LSTM networks with attention to produce sentences

Table 3: Examples of code summarization generated by each model and attention visualization of our model.

	Case 1	Case 2

Code snippet	`def _has_git():`
	` try: subprocess.check_call(`
	`  [git, -version],`
	`  stdout=subprocess.DEVNULL,`
	`  stderr=subprocess.DEVNULL)`
	` except(OSError, subprocess`
	`  .CalledProcessError):`
	`  return False`
	` else: return True`
		`def tensor3(name=None,dtype=None):`
		` if (dtype is None):`
		`  dtype=config.floatX`
		` type=CudaNdarrayType(`
		`  dtype=dtype,`
		`  broadcastable=`
		`  (False, False, False))`
		` return type(name)`

Ground truth	check if git is installed .	return a symbolic 3-d variable .
Seq2Seq	helper function to create a new figure manager instance .	yaml
Seq2Seq+Attn	return true if the user has access to the specified resource .	a decorator that returns a new class that will return a new class name .
Tree2Seq+Attn	test that validate_folders throws a foldermissingerror .	helper function for #4957 .
Hybrid2Seq+Attn	returns the number of git modules that are not installed .	return the path to the currently running server .
Hybrid2Seq+Attn+DRL	returns true if git is installed .	return a symbolic graph .
Attention visualization		

that describe C# code snippets and SQL queries. In Haije's thesis [13], the code summarization problem is modeled as a machine translation task, and some translation models such as Seq2Seq [43] and Seq2Seq with attention [4] are employed. Unlike previous studies, we take the tree structure and sequential content of source code into consideration for a better representation of code.

8.3 Deep Reinforcement Learning

Reinforcement learning [19, 45, 50], concerned with how software agents ought to take actions in an environment so as to maximize the cumulative reward, is well suited for the task of decision-making. Recently, professional-level computer Go program has been designed by Silver et al. [41] using deep neural networks and Monte Carlo Tree Search. Human-level gaming control [28] has been achieved through deep Q-learning. A visual navigation system [53] has been proposed recently based on actor-critic reinforcement learning model. Text generation can also be formulated as a decision-making problem and there have been several reinforcement learning-based works on this specific tasks, including image captioning [38], dialogue generation [20] and sentence simplification [52]. Ren et al. [38] propose an actor-critic deep reinforcement learning model with an embedding reward for image captioning. Li et al. [20] integrate a developer-defined reward with REINFORCE algorithm for dialogue generation. In this paper, we follow an actor-critic reinforcement learning framework, while our focus is on encoding the structural and sequential information of code snippets simultaneously with an attention mechanism.

9 CONCLUSION

In this paper, we first point out two issues (i.e., code representation and exposure bias) existing in traditional code summarization works. To handle these two issues, we first encode the structure and sequential content of code via AST-based LSTM and LSTM respectively. Then we add a hybrid attention layer to integrate them together. We then feed the code representation vector into a deep reinforcement learning framework, named actor-critic network. Comprehensive experiments on a real-world dataset show that our proposed model outperforms other competitive baselines and achieves state-of-the-art performance on several automatic metrics, namely BLEU, METEOR, ROUGE-L and CIDER.

ACKNOWLEDGMENTS

This work is partially supported by the Ministry of Education of China under grant of No.2017PT18, the Natural Science Foundation of China under grant of No. 61672453, 61773361, 61473273, 61602405, the WE-DOCTOR company under grant of No. 124000-11110 and the Zhejiang University Education Foundation under grant of No. K17-511120-017. This work is also supported by CCF-Tencent Open Research Fund, NSF through grants IIS-1526499, IIS-1763325, CNS-1626432, and NSFC 61672313.

REFERENCES

[1] A. V Aho, R. Sethi, and J. D Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

[2] M. Allamanis, H. Peng, and C. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[4] D. Bahdanau, K. Cho, and Y. Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] S. Banerjee and A. Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, Vol. 29. 65–72.

[6] A. V. M. Barone and R. Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.

[8] H. K. Dam, T. Tran, and T. Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[9] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM, 68–75.

[10] J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[11] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. *arXiv preprint arXiv:1603.06075* (2016).

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[13] T. Haije, B. O. K. Intelligentie, E. Gavves, and H. Heuer. 2016. Automatic Comment Generation using a Neural Translation Model. (2016).

[14] S. Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model.. In *ACL (1)*.

[16] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55.

[17] Y. Keneshloo, T. Shi, C. K Reddy, and N. Ramakrishnan. 2018. Deep Reinforcement Learning For Sequence to Sequence Models. *arXiv preprint arXiv:1805.09461* (2018).

[18] M. Kilickaya, A. Erdem, N. Ikizler-Cinbis, and E. Erdem. 2016. Re-evaluating automatic metrics for image captioning. *arXiv preprint arXiv:1612.07600* (2016).

[19] V. R. Konda and J. N. Tsitsiklis. 2000. Actor-critic algorithms. In *Advances in neural information processing systems*. 1008–1014.

[20] J. Li, W. Monroe, A. Ritter, M. Galley, J. Gao, and D. Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[21] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[22] B. P. Lientz and E. B. Swanson. 1980. Software maintenance management. (1980).

[23] C. Y. Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out* (2004).

[24] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[25] C. Maddison and D. Tarlow. 2014. Structured generative models of natural source code. In *International Conference on Machine Learning*. 649–657.

[26] R. C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[27] A. Mnih and Y. W. Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426* (2012).

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[29] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.. In *AAAI*, Vol. 2. 4.

[30] D. Movshovitz-Attias and W. W. Cohen. 2013. Natural language models for predicting programming comments. (2013).

[31] A. T. Nguyen and T. N. Nguyen. 2017. Automatic categorization with deep neural network for open-source Java projects. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 164–166.

[32] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing* 9, 5 (2016), 771–783.

[33] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 574–584.

[34] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[35] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[36] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. 2015. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969* (2015).

[37] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732* (2015).

[38] Z. Ren, X. Wang, N. Zhang, X. Lv, and L. J. Li. 2017. Deep Reinforcement Learning-Based Image Captioning with Embedding Reward. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE, 1151–1159.

[39] R. Rosenfeld. 2000. Two decades of statistical language modeling: Where do we go from here? *Proc. IEEE* 88, 8 (2000), 1270–1278.

[40] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[41] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[42] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[43] I. Sutskever, O. Vinyals, and Q. V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[44] R. S Sutton and A. G Barto. 1998. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.

[45] R. S Sutton, D. A McAllester, S. P Singh, and Y. Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. 1057–1063.

[46] K. S. Tai, R. Socher, and C. D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[47] R. Vedantam, C. Lawrence Zitnick, and D. Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4566–4575.

[48] C. J. Watkins and P. Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[49] H. H. Wei and M. Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. (2017).

[50] R. J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*. Springer, 5–32.

[51] D. Yang, A. Hussain, and C. V. Lopes. 2016. From query to usable code: An analysis of stack overflow code snippets. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 391–401.

[52] X. Zhang and M. Lapata. 2017. Sentence Simplification with Deep Reinforcement Learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 584–594.

[53] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A Gupta, L. Fei-Fei, and A. Farhadi. 2017. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 3357–3364.