Reinforcement-Learning-Guided Source Code Summarization using Hierarchical Attention

Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu

Abstract—Code summarization (aka comment generation) provides a high-level natural language description of the function performed by code, which can benefit the software maintenance, code categorization and retrieval. To the best of our knowledge, the state-of-the-art approaches follow an encoder-decoder framework which encodes source code into a hidden space and later decodes it into a natural language space. Such approaches suffer from the following drawbacks: (a) they are mainly input by representing code as a sequence of tokens while ignoring code hierarchy; (b) most of the encoders only input simple features (e.g., tokens) while ignoring the features that can help capture the correlations between comments and code; (c) the decoders are typically trained to predict subsequent words by maximizing the likelihood of subsequent ground truth words, while in real world, they are excepted to generate the entire word sequence from scratch. As a result, such drawbacks lead to inferior and inconsistent comment generation accuracy.

To address the above limitations, this paper presents a new code summarization approach using hierarchical attention network by incorporating multiple code features, including type-augmented abstract syntax trees and program control flows. Such features, along with plain code sequences, are injected into a deep reinforcement learning (DRL) framework (e.g., actor-critic network) for comment generation. Our approach assigns weights (pays "attention") to tokens and statements when constructing the code representation to reflect the hierarchical code structure under different contexts regarding code features (e.g., control flows and abstract syntax trees). Our reinforcement learning mechanism further strengthens the prediction results through the actor network and the critic network, where the actor network provides the confidence of predicting subsequent words based on the current state, and the critic network computes the reward values of all the possible extensions of the current state to provide global guidance for explorations. Eventually, we employ an advantage reward to train both networks and conduct a set of experiments on a real-world dataset. The experimental results demonstrate that our approach outperforms the baselines by around 22% to 45% in BLEU-1 and outperforms the state-of-the-art approaches by around 5% to 60% in terms of S-BLEU and C-BLEU.

Index Terms—Code summarization, hierarchical attention, reinforcement learning.

1 Introduction

In the life cycle of software development, nearly 90% of the effort is used for maintenance, and much of this effort is spent on understanding the maintenance task and related software source code via code documents [1]. In addition, it has been widely argued that code documents can benefit various software engineering techniques [2], [3], [4], [5], [6], [7], e.g., software testing [8], [9], [10], [11], [12], [13], [14], fault localization [15], [16], [17], program repair [18], [19], [20].

- *W. Wang is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong, 518055, P. R. China and the School of Software, University of Technology, Sydney, NSW 2008 Australia.*
  E-mail: 11760006@mail.sustech.edu.cn.
- *Y. Zhang is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong, 518055, P. R. China.*
  E-mail: zhangyq@sustech.edu.cn.
- *Y. Sui, and G. Xu are with the School of Software, University of Technology, Sydney, NSW 2008 Austrialia.*
  E-mail: Yulei.Sui@uts.edu.au, Guandong.Xu@uts.edu.au.
- *Y. Wan, Z. Zhao and J. Wu are with Zhejiang University, Hangzhou, Zhejiang, P. R. China.*
  E-mail: wanyao1992@gmail.com, Zhaozhou@zju.edu.cn, wujian2000@zju.edu.an.
- *P. Yu is with the University of Illinois at Chicago, Illinois, USA.*
  E-mail: psyu@uic.edu.
- *Corresponding Author: Yuqun Zhang, Guandong Xu and Yulei Sui*

Manuscript received XXX, XX, 2019; revised XXX, XX, 2019.

Thus, it is essential for documentation to provide a high-level description of the task performed by code for software maintenance. Even though various techniques have been developed to facilitate programmers during software implementation and testing, documenting code with comments remains a labour-intensive task [21], [22], [23]. In fact, few real-world software projects can adequately document code to reduce future maintenance costs [24], [25]. It is even challenging and time-consuming for a novice programmer to write good comments for code. Typically, good comments should have the following characteristics: (1) *Correctness*. The comments should correctly clarify the intent of code. (2) *Fluency*. The comments should be fluent, so that they can be easily read and understood by maintainers. (3) *Consistency*. The comments should follow a standard style/format for better code reading. To this end, code summarization is proposed to comprehend code and automatically generate descriptions directly from code. Summarizing code can also be viewed as a form of document expansion , where a successful code summary can not only benefit the maintenance of source code [26], [27], but also be used to improve the performance of code search using natural language queries [28], [29] and code categorization [30].

Existing Efforts. Recent research has made some progress towards automatic generation of natural language descriptions of software. As far as we know, most of the existing code summarization approaches learn the semantic representation of code based on statistical language models [26], [31],

and then generate comments based on templates or rules [32]. With the development of deep learning, some neural translation models [27], [33], [34] have also been introduced for code summarization, which mainly follow an encoder-decoder framework. They generally employ recurrent neural networks (RNNs) [35] to encode the code snippets into a hidden space and utilize another RNN to decode that hidden space to coherent sentences. Given the predecessor words and the ground truth, these models are typically trained to maximize the likelihood of subsequent words.

Limitations and Insights. Based on our observation, the existing approaches suffer from the following three limitations: (1) Most of the existing approaches input code as plain texts that are composed of tokens (i.e., variables, operations, etc.) directly without considering the code hierarchy (e.g., tokens forming a statement and statements forming a function) which can provide more comprehensive representation by differentiating tokens under different contexts for comment generation. (2) Most of the existing approaches [26], [33], [36] utilize simple sequential features, such as token sequence, for code representation, while some code features (e.g., control flow graphs (CFGs), Abstract Syntax Tree (AST), and types of program variables) that can help capture the correlations between comments and programs remain unexplored. (3) The existing training approaches, also termed "teacher-forcing" models, suffer from the *exposure bias* issue which occurs as the ground truth is unavailable during testing stage and the previously generated words from the trained model are used to predict subsequent words [37] so that the model is only trained based on ground truth context and is not exposed to its own errors [38].

Our Solutions. To tackle the aforementioned problems, we present a new hierarchical-attention-based learning approach by utilizing multiple structural code features (including control flow graph and AST) to reflect the code hierarchy, where a two-layer attention network (a token layer and a statement layer) is established for an effective code representation that differentiates tokens under different contexts for comment generation.

In our previous work [39], we proposed a deep reinforcement learning-based approach which draws on the insights of deep reinforcement learning to alleviate the exposure bias issue by integrating exploration and exploitation into the whole framework (addressing limitation (3)). It first encodes the structure and sequential content of code via an AST-based LSTM and a regular LSTM respectively. Next, the resulting code representation vector is fed into a deep reinforcement learning framework, namely actor-critic network. Instead of learning a sequential recurrent model to greedily look for the subsequent correct word, we utilize an actor network and a critic network to jointly determine the subsequent optimal word at each time step. In particular, the actor network provides the confidence of predicting the subsequent word according to the current state. The critic network, on the other hand, computes the reward values of all the possible extensions of the current state. As a result, our approach successfully collects the appropriate words that are less likely to be identified by using the actor network only. To learn these two networks more efficiently, our approach is initialized by pretraining an actor network using standard supervised learning with cross entropy loss, and pretraining a critic network with mean square loss. Accordingly, we update the actor and critic networks based on the advantage reward composed of BLEU metric via policy gradient.

In this paper, we further extend the previous approach to improve the efficacy by replacing the AST-based tree structure representation with type-augmented AST sequence and complementing the code representation with control flows (addressing limitation (2)). Moreover, we adopt the hierarchical attention network ((HAN) to encode sequence of different code representations (addressing limitation (1)). Specifically, first, in our previous work, AST is used to reflect the feature information of the code. However, constructing AST-based LSTM is time-consuming. Hence, we extract both unstructured and structured information (e.g., control flows and type-augmented AST) from code to efficiently represent code. The unstructured information is obtained directly by transforming code to plain text following the existing approach [27], while the structured information is represented by the type-augmented abstract syntax tree [40] sequence which is a syntactic code representation widely used in compilers and its corresponding control flow graph. Second, in our previous work, we transform code to be plain text that is composed of tokens directly, while ignoring the code hierarchy. Thus, we adopt the hierarchical attention network to encode a code sequence to effectively represent the code hierarchy by fusing different representations of the code into a low-dimensional and compact feature space. This hierarchical attention network assigns weights (pays "attention") to individual tokens and statements regarding different code representations.

Framework Overview. Figure 1 gives an overview framework of our reinforcement-learning-guided comment generation approach via two-layer attention network, which includes an offline training stage and an online testing (summarization) stage. In the offline training stage, we prepare a large-scale corpus of annotated $< code, comment >$ pairs. Specifically, first, we used three sequences: x^{TXT}, x^{AST} and x^{CFG}, to represent code at both unstructured level (plain code sequence) and structured level (type-augmented ASTs and control flows) (Figure 1(a)); next, we use the hierarchical attention network (Figure 1(b)) to encode these three representations and integrate them. At last, the annotated pairs are injected into our proposed deep reinforcement learning model (Figure 1(c)) for training. Given the resulting trained actor network and a code snippet, its corresponding comment can be generated. Note that in this paper, we only deal with function-level summarization.

Contributions. The main contributions of this paper are as follows:

- **New idea**. We propose a deep reinforcement learning framework—actor-critic network for comment generation which copes with the exposure bias issue existing in most traditional maximum likelihood-based approaches.
- **Extensive algorithms.** This paper presents the first hierarchical-attention-based learning approach for code summarization by utilizing multiple code features (i.e., plain text, type-augmented AST and CFG) to reflect code hierarchy (tokens forming a statement, statements forming a function) by supporting a two-layer attention network at both token level and statement level to pro-

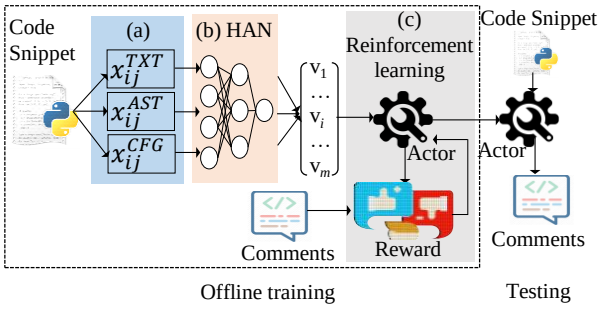
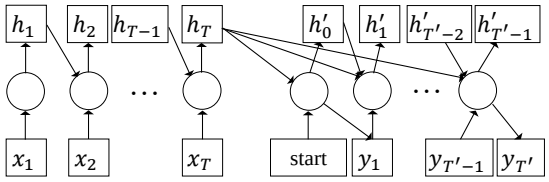


Figure 1: An overview framework of our proposed approach (HAN is the Hierarchical Attention Network).



(a) The architecture of encoder in RNN (b) The architecture of decoder in RNN

Figure 2: The structure of recurrent neural network (RNN).

vide an effective code representation that differentiates tokens under different contexts for comment generation.
- **Evaluation.** We validate our proposed approach on a real-world dataset of 108,726 Python code snippets used in our previous work and 20,000 more Python code snippets for the testing. Moreover, compared with the previous work, we implemented more existing approaches for performance comparison. The overall experimental results demonstrate that our approach outperforms the baselines in terms of comment generation accuracy (from around 22% to 45% in BLEU-1) and outperforms the state-of-the-art approaches by around 5% to 60% in terms of both Sentence-BLEU and Corpus-BLEU.

The remainder of this paper is organized as follows. Section 2 illustrates some preliminary background knowledge. An illustrative example is given in Section 3. The details of our proposed approach are elaborated in Section 4. Section 5 demonstrates the experimental results and analysis. Threats to validity are indicated in Section 6. Section 7 reviews the related work. We conclude the paper in Section 8.

2 Preliminaries

In this section, we first present some preliminary background knowledge about text generation, including language model, RNN encoder-decoder model, and the reinforcement learning for decoding. Firstly, we introduce some basic notations and terminologies. Let $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$ denote the code sequence of one function, where x_i represents a token of the code, e.g., "def", "fact", or "i" in a Python statement "def fact(i):". Let $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$ denote a sequence of the generated comments, where $|\cdot|$ denotes the sequence length. Let T denote the maximum step of decoding in the encoder-decoder framework. We use notation $\mathbf{y}_{l\ldots m}$ to represent $y_l, \ldots, y_m$ and $\mathcal{D} = \{(\mathbf{x}_N, \mathbf{y}_N)\}$ as the training dataset, where N is the size of training set.

2.1 Language Model

Language model computes occurrence probability of the words in a particular sequence [41]. The probability of a sequence including T words: $\mathbf{y}_{1\ldots T}$ is denoted as $p(y_{1:T})$ which is usually computed based on the conditional probability from a window of n predecessor words, aka n-gram [42], as shown in Equation 1.

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

Such n-gram approach suffers from apparent limitations [43], [44]. For example, the n-gram model is derived only from the frequency counts and leads to inferior performance when confronted with the tokens that have not frequently appeared before.

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 distance. The associated neural network includes three layers, i.e., an input layer which maps each word x_t to a vector, a recurrent hidden layer which recurrently computes and updates a hidden state h_t after reading x_t, and an output layer which estimates the probabilities of the subsequent words given the current hidden state. In particular, the neural network reads the words in the sentence one by one, and predicts the possible subsequent word at each time. At time t, it estimates the probability of the subsequent word $p(y_{t+1}|y_{1:t})$ by the following steps: (1) the current word y_t is mapped to a vector by the input layer; (2) it generates the hidden state (the values in the hidden layer) $\mathbf{h}_t$ at time t according to the previous hidden state $\mathbf{h}_{t-1}$ and the current input x_t: $\mathbf{h}_t = f(\mathbf{h}_{t-1}, w(x_t))$, where w refers to the parameters of the networks.

(3) the $p(y_{t+1}|y_{1:t})$ is predicted according to the current hidden state $\mathbf{h}_t$: $p(y_{t+1}|y_{1:t}) = g(\mathbf{h}_t)$, where g is a stochastic output layer (e.g., a softmax for discrete outputs) that generates output tokens.

2.2 RNN Encoder-Decoder Model

RNN (Recurrent Neural Network) encoder-decoder, as shown in Figure 2, 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 an RNN, while the decoder uses another RNN to generate one word y_t at a time in the target space.

2.2.1 Encoder

A hidden state in an RNN encoder (Figure 2(a)) is a fixed-length vector. At the time t, the encoder computes the hidden state $\mathbf{h}_t$ as shown in Equation 2.

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, w(\mathbf{x}_t)) \quad (2)$$

Here, $\mathbf{h}_{t-1}$ denotes the hidden state at last step, $\mathbf{x}_t$ denotes the input at step t, and f is the hidden layer. The last symbol of $\mathbf{x}$ should be an end-of-sequence ($< eos >$) symbol which notifies the encoder to terminate and output the final hidden state $\mathbf{h}_T$, which is used as a vector representation of $\mathbf{x}_{1:T}$.

RNN has two shortcomings: gradient vanishing and gradient exploding which refer to the large decrease and increase in the norm of the gradient during training. To alleviate this problem, the long short-term Memory (LSTM) [45] technology with a gate mechanism is proposed to determine the information accumulation, where the input gate, forget gate and output gate control the input, forgt and output part of the entire network through weights and activation function. In this paper, we choose LSTM as our encoder. At time t, the hidden state is updated as follows,

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}^{(i)}\mathbf{x}_t + \mathbf{U}^{(i)}\mathbf{h}_{t-1} + \mathbf{b}^{(i)}) \\
\mathbf{f}_t &= \sigma(\mathbf{W}^{(f)}\mathbf{x}_t + \mathbf{U}^{(f)}\mathbf{h}_{t-1} + \mathbf{b}^{(f)}) \\
\mathbf{a}_t &= \tanh(\mathbf{W}^{(a)}\mathbf{x}_t + \mathbf{U}^{(a)}\mathbf{h}_{t-1} + \mathbf{b}^{(a)}) \\
\mathbf{c}_t &= \mathbf{i}_t \odot \mathbf{a}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} \\
\mathbf{o}_t &= \sigma(\mathbf{W}^{(o)}\mathbf{x}_t + \mathbf{U}^{(o)}\mathbf{h}_{t-1} + \mathbf{b}^{(o)}) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned} \tag{3}$$

where $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$ and $\mathbf{a}_t$ denote an input gate, a forget gate, an output gate, and a intermediate parameter respectively for updating the memory cell c_t. $\mathbf{W}^{(\cdot)}$ and $\mathbf{U}^{(\cdot)}$ are weight matrices, $\mathbf{b}^{(\cdot)}$ is a bias vector, and $\mathbf{x}_t$ is the word embedding of the tth node. $\sigma(\cdot)$ is the logistic function, and the operator $\odot$ denotes element-wise multiplication between vectors.

2.2.2 Decoder

The output of the decoder (Figure 2(b)) is the target sequence $\mathbf{y} = (y_1, \cdots, y_{T'})$. The decoder is initialized to input a $< start >$ symbol denoting the beginning of the target sequence. At time t, the decoder computes the conditional distribution of the subsequent symbol y_{t+1} based on the hidden state $\mathbf{h}_t$: $p(y_{t+1}|y_t) = g(\mathbf{h}_t)$, where g is a stochastic output layer.

2.2.3 Training Goal

The encoder and decoder networks are jointly trained to maximize the following objective,

$$\max_{\theta} \mathcal{L}(\theta) = \max_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \log p(\mathbf{y}|\mathbf{x};\theta) \tag{4}$$

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 from the exposure bias issue. Accordingly, we introduce the reinforcement learning technique for better decoding.

2.3 Reinforcement Learning for Better Decoding

Reinforcement learning [46] interacts with the environment and learns the optimal policy from the reward signal, which can potentially solve the exposure bias problem introduced by the maximum likelihood approaches which is used to train the RNN model. Specifically in the inference stage, a typical RNN model generates a sequence iteratively and predicts next token conditioned on its previously predicted ones that may never be observed in the training data [47]. Such a discrepancy between training and inference can become cumulative along with the sequence and thus prominent as the length of sequence increases. While in the reinforcement-learning-based framework, the reward, other than the probability of the generated sequence, is calculated to give feedback to train the model to alleviate such exposure bias problem. Accordingly, 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 t consists of the code snippets $\mathbf{x}$ and the predicted words $y_0, y_1, \ldots, y_t$. The action space is defined as the dictionary $\mathcal{Y}$ where the words are drawn, *i.e.*, $y_t \subset \mathcal{Y}$. Correspondingly, the state transition function P is defined as $\mathbf{s}_{t+1} = \{\mathbf{s}_t, y_t\}$, where the action (word) y_t becomes a part of the subsequent state $\mathbf{s}_{t+1}$ and the reward r_{t+1} is received. The objective of the generation process is to find a policy that maximizes the expected reward of the generated sentence sampled from the model's policy, as shown in Equation 5,

$$\max_{\theta} \mathcal{L}(\theta) = \max_{\theta} \mathbb{E}_{\substack{\mathbf{x}\sim\mathcal{D} \\ \hat{\mathbf{y}}\sim P_\theta(\cdot|\mathbf{x})}} [R(\hat{\mathbf{y}}, \mathbf{x})] \tag{5}$$

where θ is the policy parameter 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:

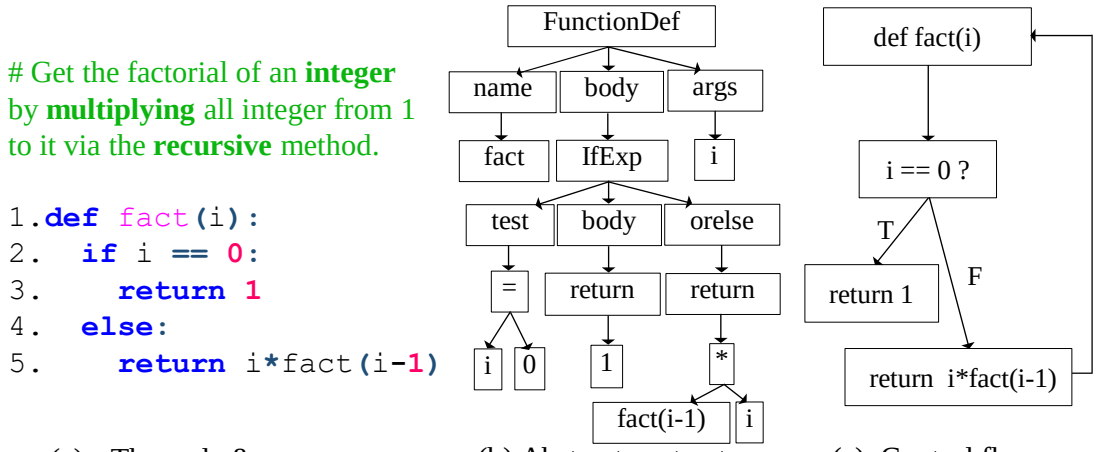
- Given a code snippet $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$, our goal is to find a policy that generates a sequence of words $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$ from dictionary $\mathcal{Y}$ with the objective of maximizing the expected reward.

To learn the policy, many approaches have been proposed, which are mainly categorized into two classes [48]: (1) the policy-based approaches (e.g., Policy gradients [49]) which optimize the policy directly via policy gradient and (2) the value-based approaches (e.g., Q-learning [50]) which learn the Q-function, and in each time the agent selects the action with the highest Q-value. It has been verified that the policy-based approaches may suffer from a variance issue and the value-based approaches may suffer from a bias issue [51]. To combine the advantages of both policy- and value-based approaches, the Actor-Critic learning approach is proposed [52]. In particular, the actor chooses an action according to the probability of each action and the critic assigns the value to the chosen action, which speeds up the learning process for the original policy-based approaches.

3 Illustrative Example

In this section, we use a Python code snippet as our illustrative example.

Figure 3(a) shows a simple Python code example which aims to obtain the factorial of an integer via a recursive function `fact`. Figure 3(b) is the AST of the code in Figure 3(a). Figure 3(c) shows its inter-procedural control flow graph which represents program execution order. The ideal comments (green) of this code is given in Figure 3(a). It can be indicated that the semantics of the three highlighted words can be precisely captured by different code representations,



(a) The code & summary (b) Abstract syntax tree (c) Control flow

Figure 3: (a) Code snippet and the corresponding summary. (b) The AST sequence of the code obtained by the *ast* module of Python. (c) The inter-procedural control flow of the code.

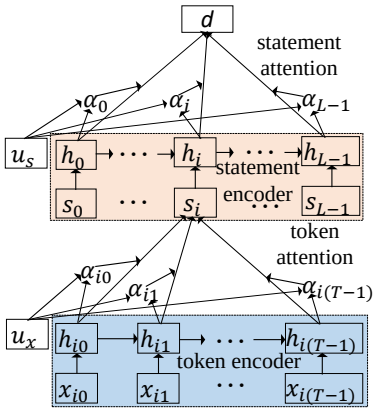


Figure 4: The architecture of a two-layer hierarchical attention network.

e.g., plain text (for **multiplying**), type-augmented AST (for **integer**) and CFG (for **recursive**).

The order of tokens and statements can vary depending on different code representations. In this paper, we use the three unstructured and structured information of code, i.e., plain text, AST and CFG. For example, based on plain text, the token after "if" in Figure 3 (a) is "i" followed by "==". Based on the AST sequence represented as: {stmt = FunctionDef(identifier fact, arguments i, stmt body); body = IfExp(expr test, expr body, expr orelse); ...}, there are three tokens (test, body and orelse) following "IfExp" with "=" following "test", "return" following "body", and "orelse", as shown in Figure 3 (b). After the token "1" in the last statement at line 5, there is no token left according to plain text. However, based on CFG, the subsequent token is "def" at the beginning of `fact` due to the recursive function call.

From Figure 3(a), we can observe that tokens "def", "fact", and "i" form statement 1, while statements 1 to 5 form the entire function. Such hierarchy is captured by a two-layer attention network (including one token layer and one statement layer), as shown in Figure 4, where the bottom layer encodes each token x_{it} of statement s_i to produce a vector of this statement, which is later injected into the top-layer attention network along with the other statements to obtain a final vector to represent their associated function. Note that in Figure 4, α_i and α_{it} represent the weights of the ith statement and the tth token of statement s_i respectively, which are inferred during reinforcement learning.

By utilizing the three code representations and the hierarchical attention network, our approach produces three different vectors with different token and statement sequences. Finally, the three vectors are concatenated to produce the final code representation for precisely capturing the relations between tokens and relations between statements.

4 The DRL-guided Code Summarization via Hierarchical Attention Network

In this section, we introduce the details of our proposed DRL (Deep Reinforcement Learning)-guided code summarization approach via hierarchical attention network with its architecture shown in Figure 5. Our approach follows the actor-critic framework [53], which has been successfully adopted in the decision-making scenarios such as AlphaGo [54]. Specifically, we split the framework into four submodules: (a) code representations which are used to explain the unstructured and structural information of a program; (b) hybrid hierarchical attention network which is used to encode the code representations into vectors in the hidden space; (c) text generation which is a LSTM-based generation network to generate the subsequent words based on predecessor words; and (d) the critic network which is used to evaluate the quality of the generated word.

4.1 Source Code Representations

For the identifiers in source code, we tokenize and split them by a set of symbols, i.e., {. , " ' _ ( ) { } : ! - (space) }. Next, all the resulting tokens are changed to lowercase letter. Furthermore, we embed all the obtained tokens to vectors by *Word2Vec*() provided by Python library *genism* [55], where similar to [27], [56], [57], the undefined tokens are dealt as the unknown words.

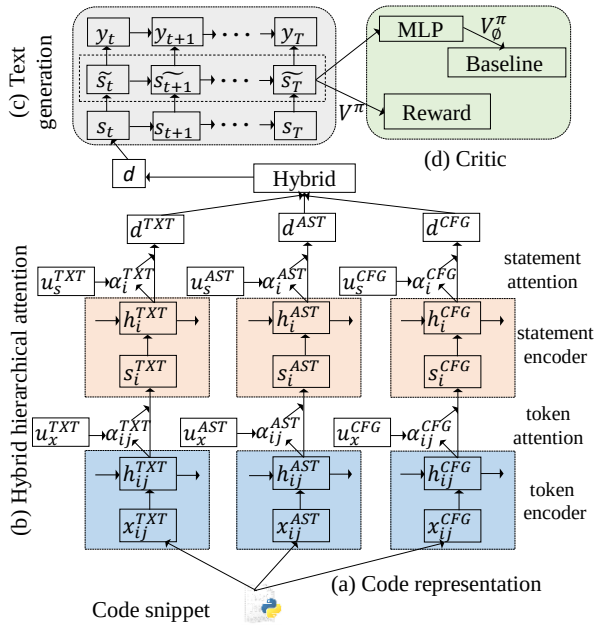


Figure 5: An overview of our proposed reinforcement-learning-guided code summarization via hierarchical attention network: (a) x_{ij}^{TXT}, x_{ij}^{AST} and x_{ij}^{CFG} represent the jth token in the ith statement of the lexical level representation, the type-augmented AST representation, and the control flow representation of the code respectively; (b) the LSTM-based hierarchical attention network is used to encode the three code representations into vectors: d_t^{TXT}, d_t^{AST} and d_t^{CFG} respectively, followed by a hybrid layer which is used to integrate these three vectors; (c) the LSTM-based decoder is used to generate the summary of the code; (d) Given a state s_t, the critic network evaluates its value (baseline) with the advantage defined as $|baseline - reward|$.

Based on the resulting tokens, we introduce the following three code representations: plain text, type-augmented abstract syntax tree, and control flow graph.

4.1.1 Plain Text

The key insight into the lexical level representation of code is that comments are always extracted from the lexical items of source code, such as the function name, variable name and so on.

4.1.2 Type-augmented Abstract Syntax Tree

When executing a program, a compiler decomposes a program into constituents and produces intermediate code according to the syntax of the programming language, such as AST [58]. In this paper, first, we obtain the AST sequence by the *ast* module [59] of Python as one syntactic-level representation of the code. Next, to augment the derived AST sequence with additional type information, we propose to abstract the type information of the tokens and integrate them with the AST sequence of the code. For example, in Figure 3 (a), line 2 is represented as "if integer i == integer 1" by annotating "integer" type to variable "1".

4.1.3 Control Flow Graph

Since different code representations reflect different latent code features, we extract the control flow graph (CFG), which is another type of intermediate code often used in compiler, as another syntactic level representation of the code. In particular, each node on CFG represents a statement consisting of a sequence of tokens and each edge connecting two nodes denotes the program's control flow. The control flow graph is obtained by following the *ast* module [59] and [60] to and then traverse the obtained graph in depth-first order to obtain the control flow sequence .

4.2 Hybrid Hierarchical Attention Network

Each code part makes its own contribution to the final output of comments. Specifically, first, the importance of tokens and statements are highly context dependent, i.e., the same token or statement may be differentially important in different context. Next, code essentially has a hierarchy (tokens forming statements and statements forming functions). Therefore, the hierarchical attention network [61], which has been successfully used in natural language processing, is applied to allow the approach to assign weights (pay "attention") to individual tokens and statements respectively when constructing the code representations. Attention not only often results in better performance, but also provides insights into the correlations between tokens/statements and the corresponding summary, which benefits in generating high-quality comments [62], [63].

In this paper, we apply a two-layer attention network (a token layer and a statement layer), as shown in Figure 5 (b). This network consists of four parts: a token sequence encoder, a token-level attention layer, a statement encoder, and a statement-level attention layer. Assuming d^{TXT}, d^{AST}, and d^{CFG} are the vectors deducted by encoding the three code representations i.e., plain text, AST, and CFG. As a result, they are merged into one hybrid vector d to represent code. The details of such network are demonstrated as follows.

Token Encoder. Given a statement s_i with tokens $x_{i0}, ..., x_{iT_i-1}$, where T_i is the total number of tokens in s_i, we first embed all the tokens to vectors through an embedding matrix W_i, i.e., $v_{it} = W_i x_{it}$. Next, we use an LSTM to obtain token annotations by reading statement s_i from x_{i0} to x_{iT_i-1}, as shown in Equation 6.

$$\begin{aligned} \mathbf{v}_{it} &= \mathbf{W}_i \mathbf{x}_{it}, t \in [0, T_i) \\ \mathbf{h}_{it} &= lstm(\mathbf{v}_{it}), t \in [0, T_i) \end{aligned} \tag{6}$$

Token Attention. Not all tokens contribute equally to the semantic representation of the statement. For example, in Figure 6, tokens "number" and "str" are essentially more important than tokens "def" in statement "def check_number_exist(str):" because there are words "numbers" and "string" in the comment of this code snippet. Hence, we introduce the attention mechanism to extract the tokens that are more important to the semantics of a statement and aggregate the representation of those informative tokens to form a statement vector as shown in Equation 7.

```
# Check if there are numbers in a string.

def check_number_exist(str):
    has_number = False
    for c in str:
        if c.isnumeric():
            has_number = True
            break
    return has_number
```

Figure 6: Code example of different tokens contributing differently for comment generation.

$$\begin{aligned} \mathbf{u}_{it} &= tanh(\mathbf{W}_x\mathbf{h}_{it} + \mathbf{b}_x) \\ \alpha_{it} &= \frac{exp(\mathbf{u}_{it}^T\mathbf{u}_x)}{\Sigma_T exp(\mathbf{u}_{it}^T\mathbf{u}_x)} \\ \mathbf{s}_i &= \Sigma_T \alpha_{it}\mathbf{h}_{it} \end{aligned} \tag{7}$$

Here, W_x is the weight matrix, b_x is a bias vector, α_{it} denotes the contribution (attention) of token x_{it} to statement s_i, and u_x is the token-level context vector which is used for the high-level representation of each statement in terms of tokens. In particular, u_x is randomly initialized and gradually learned during the training process.

Statement Encoder. Given the statement vector s_i, we can obtain a function vector in a similar manner to tokens. We use an LSTM to encode the statements as follows.

$$\mathbf{h}_i = lstm(\mathbf{s}_i), i \in [0, L) \tag{8}$$

Here, L is the total number of the statements included in one function.

Statement Attention. To reward the statements that are more semantically important to the associated function for the summarization task, we again use attention network and introduce a statement level function vector u_s which is used to measure the importance of the statement as follows.

$$\begin{aligned} \mathbf{u}_i &= tanh(\mathbf{W}_s\mathbf{h}_i + \mathbf{b}_s) \\ \alpha_i &= \frac{exp(\mathbf{u}_i^T\mathbf{u}_s)}{\Sigma_L exp(\mathbf{u}_i^T\mathbf{u}_s)} \\ \mathbf{d}^c &= \Sigma_L \alpha_i \mathbf{h}_i \end{aligned} \tag{9}$$

Here, W_s is the weight matrix, b_s is a bias vector, and α_i denotes the contribution (attention) of statement s_i to the final vector $\mathbf{d}^c$.

Hybrid Representation of Source Code. To integrate the structural context vector (i.e., AST and CFG representations) and the unstructural textual vector (i.e., plain text representation), we concatenate them firstly and later feed them into a one-layer linear network: $\mathbf{d} = \mathbf{W_d}[\mathbf{d}^{TXT}; \mathbf{d}^{AST}; \mathbf{d}^{CFG}] + \mathbf{b_d}$, where $\mathbf{d}$ is the hybrid representation of code, $[\mathbf{d}^{TXT}; \mathbf{d}^{AST}; \mathbf{d}^{CFG}]$ is the concatenation of $\mathbf{d}^{TXT}$, $\mathbf{d}^{AST}$, and $\mathbf{d}^{CFG}$, and $\mathbf{b_d}$ is a bias vector. The context vector is then used for word prediction by placing an additional hidden layer: $\widetilde{\mathbf{s}_t} = \tanh(\mathbf{W}_c\mathbf{s}_t + \mathbf{b}_d)$, where s_t is hidden state of the encoding process. To be specific, initially in the decoding process, s_0 is assigned to be $\mathbf{d}$. Accordingly, the state s_t is updated at decoding step t.

4.3 Text Generation

After obtaining the representation of code snippet from the hierarchical attention network and the hybrid layer, we decode it into a natural language comments. Here we describe how we generate a comment from the hidden space.

For the decoding, since we design a hierarchical and multi-dimensional input, we decide to adopt the Input-feeding attention mechanism [64] to predict the tth 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 the tth word y_t, we can obtain the following equation:

$$p_\pi(y_t|\mathbf{s}_t) = softmax(\mathbf{W}_s\widetilde{\mathbf{s}_t} + \mathbf{b}_s) \tag{10}$$

4.4 Critic Network

Unlike traditional encoder-decoder frameworks [65] that generate comments directly via maximizing likelihood of subsequent words given the ground truth word, we generate comments by iteratively optimizing the evaluation metrics e.g., BLEU [66], in a reinforcement learning manner. Specifically, we apply a critic network to approximate the value of a generated action at time t to issue a feedback to tune the network iteratively. Different from the actor network, this critic network outputs a single value instead of a probability distribution on each decoding step.

To illustrate, given the generated comments and the reward function r, the value function V is defined to predict the total reward from the state s_t at time t, which is formulated as follows,

$$V(\mathbf{s}_t) = \mathbb{E}_{\substack{\mathbf{s}_{t+1:T}, \\ y_{t:T}}}\left[\sum_{l=0}^{T-t} r_{t+l}|y_{t+1}, \cdots, y_T, \mathbf{h}\right] \tag{11}$$

where T is the max step of decoding and $\mathbf{h}$ is the representation of code snippet.

By applying the reward function, we can obtain an evaluation score (e.g., BLEU) when the generation process of the comment sequences is completed. Such process is terminated when the associated step exceeds the max-step T or generates the end-of-sequence (EOS) token. For instance, a BLEU-based reward function can be calculated as

$$r = exp(\frac{1}{N} * \sum_{i=1}^{N} logp_n) \tag{12}$$

where $p_n = \frac{\sum_{n-gram \in c} count(n-gram)}{\sum_{n-gram' \in c'} count(n-gram')}$, c is the generated comment and $c^{'}$ is the ground truth.

4.5 Model Training

For actor network, the training objective 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 the actor network. Defining policy as the probability of a generated comment, we adopt the policy gradient approach to optimize the policy directly, which is widely used in reinforcement learning.

The critic network attempts to minimize the following loss function,

Algorithm 1 Actor-Critic training for code summarization.

1: Initialize actor network $p(y_{t+1|s_t})$ and critic network $V(s_t)$ with random weights θ and ϕ;
2: Pre-train the actor network to predict ground truth y_t given $\{y_1, \cdots, y_{t-1}\}$ by Eq. 4;
3: Pre-train the critic network to estimate $V(s_t)$ with fixed actor network;
4: **for** $t = 1 \rightarrow T$ **do**
5: Receive a random example, and generate the comment sequence $\{y_1, \cdots, y_T\}$ according to current actor network;
6: Calculate the reward value according to Eq. 12;
7: Update critic network weights ϕ;
8: Update actor network weights θ;

$$\mathcal{L}(\phi) = \frac{1}{2} \left\| V(\mathbf{s}_t) - V_\phi(\mathbf{s}_t) \right\|^2 \tag{13}$$

where $V(\mathbf{s}_t)$ is the target value (calculated by the value function based on the ground truth), $V_\phi(\mathbf{s}_t)$ is the value predicted by the critic network based on the generated comments with its parameter set ϕ. Eventually, the training for code summarization is completed after $\mathcal{L}(\phi)$ converges.

Denoting all the parameters as $\Theta = \{\theta, \phi\}$, the total loss of our model can be represented as $\mathcal{L}(\Theta) = \mathcal{L}(\theta) + \mathcal{L}(\phi)$. We employ stochastic gradient descend with the diagonal variant of AdaGrad [67] to tune the parameters for optimizing the code summarization model.

5 Experiments and Analysis

The goal of our evaluation is to show that our reinforcement-learning-guided approach using hierarchical attention network is more effective in generating comments than baselines (e.g., without hierarchical attention network and/or without reinforcement-learning) and state-of-the-art approaches. Our evaluation focus on the following research questions.

- **RQ1.** What is the effectiveness of our approach in generating comments and what are the results under different configuration settings?
- **RQ2.** What is the time consumption and performance trend regarding the increment of training epochs during model training?
- **RQ3.** What is the performance of our proposed approach on the datasets with different code or comment lengths?
- **RQ4.** How does our approach perform compared with other existing code summarization approaches?
- **RQ5.** How do we extensively evaluate our approach other than only relying on NLP-specific metrics?

5.1 Dataset Preparation

To evaluate the performance of our proposed approach, we use the Python dataset used in our previous work [39], which is obtained from a popular open source project hosting platform GitHub [68] and processed by Barone et. al [69]. In particular, we remove the low-quality <code, comment> pairs, e.g., comments with massive misspelling words, broken sentences, from the original dataset. As a result, the derived dataset consists of around 108K pairs.

In addition to our previous work [39], we also collect another 20K testing pairs to evaluate how our approach and baselines perform in diverse datasets to evaluate their universal applicability. In particular, to ensure that the extended testing data do not overlap the original training dataset, we first collect the projects with 80 to 100 *stars* for deriving the extended testing dataset while the original training dataset are made by the projects with more than 100 *stars*. Next, we sort the collected projects by the their number of *forks* and select the top 20k pairs accordingly. Eventually, our Python dataset contains 128K code-comment pairs in total, where the vocabulary size of code and comment is 50,400 and 31,350, respectively. Similar to [27], [56], we shuffle the original dataset and use the first 80% for training and validation and the remaining 20% for testing. Moreover, we use 10-fold cross-validation to evaluate the performance of their proposed approach. Specifically, we split the training and validation data into 10-fold, where each time we utilize 10% of the data for validation and the rest 90% for training. Then we average the testing results out of the 10 times of execution.

We also adopt the Java project dataset in [70] to evaluate the cross-language performance of our approach. Specially, we select the same number of training data, validation data and testing date as our python dataset from the original dataset in [70] in a top-down manner.

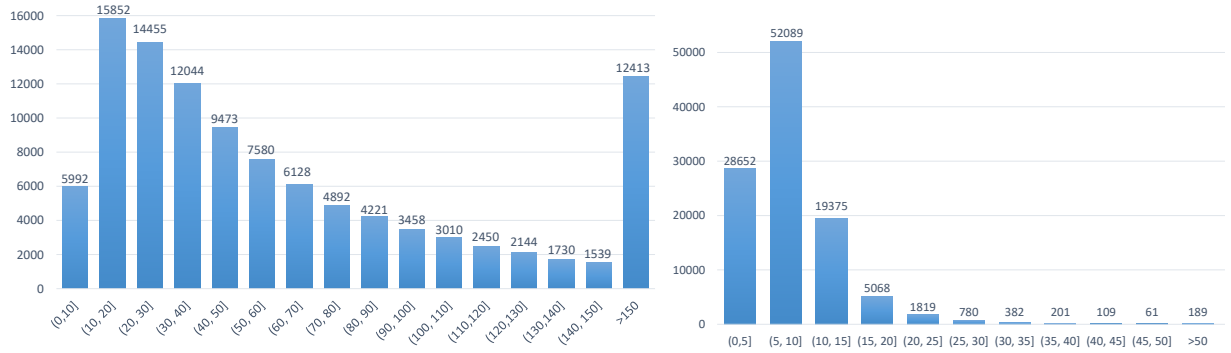
We have conducted statistics analysis for the source code and comment out of our adopted Python dataset based on massive GitHub projects as shown in Figures 7 and 8. Figure 7 shows the length distributions of code and comment. From Figure 7 (a), we can find that the lengths of most code snippets are located between 10 to 80 tokens. From Figure 7(b), we can notice that the length of nearly all the comments are between 5 and 40. This reveals that the comment sequence to be generated is not too long. Moreover, Figure 8 shows the token number and statement number distribution in the collected code snippets of our dataset where Figure 8 (a) shows the token number distribution in each statement and Figure 8 (b) shows the statement number distribution in each function. From this figure, we can observe that the token number in each statement mainly ranges from 1 to 15, and the statement number in each function mainly ranges from 2 to 25.

5.2 Evaluation Metrics

We evaluate the performance of our proposed approach based on three widely-used evaluation metrics in the area of NLP, especially for the text generation task, i.e., BLEU [66], METEOR [71] and ROUGE-L [72]. Since code summarization is a special type of text generation with natural language as the output, we utilize these evaluation metrics to evaluate the quality of the generated comments.

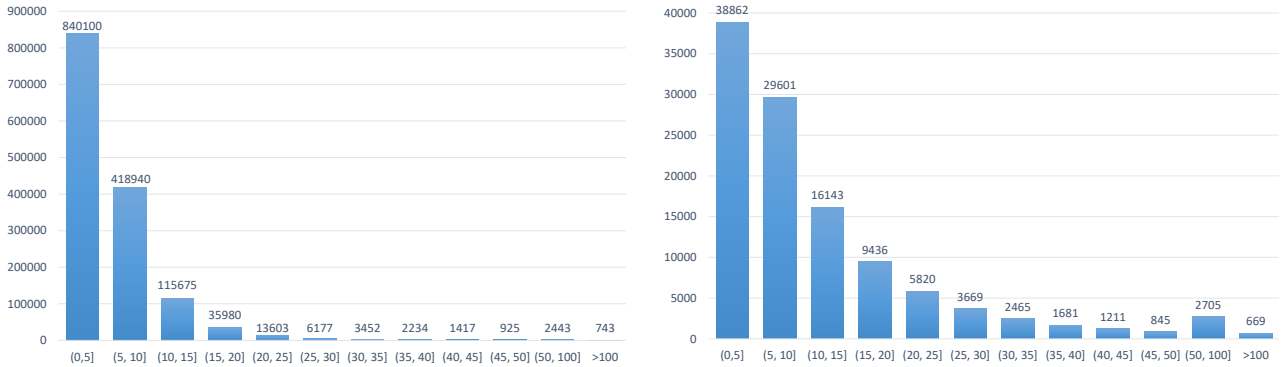
BLEU is the most common metric adopted in text generation [73], [74], [75], [76], [77], [78], [79] which measures the average n-gram precision on a set of reference sentences, with a penalty for short sentences. BLEU is calculated as:

$$BLEU = exp(\frac{1}{N} * \sum_{i=1}^{N} logp_n), \tag{14}$$



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

Figure 7: Length distribution of testing data.



(a) Token number distribution in each statement. (b) Statement number distribution in each function.

Figure 8: Statistic analyses of the source code.

where $p_n = \frac{\sum_{n-gram \epsilon c} count(n-gram)}{\sum_{n-gram' \epsilon c'} count(n-gram')}$, c is the generated comment and $c^{'}$ is the ground truth. In this paper, we adopt the BLEU-N metrics as in our previous paper. Moreover, we extend our adoption of BLEU metrics by including both the sentence-level BLEU (S-BLEU) and corpus-level BLEU (C-BLEU) for the performance comparison between our approach and state-of-the-art approaches. In particular, S-BLEU calculates the BLEU score between each generated comment and the ground truth and then calculates the average of all the scores. For S-BLEU, we adopt the add-k smoothing, for which we define k as $1e-15$ such that the count of n-gram cannot be 0. C-BLEU, on the other hand, computes the BLEU score in the corpus level.

METEOR is a recall-oriented metric which evaluates how well the results capture content from the references via computing recall by stemming and synonymy matching. It is computed as:

$$METEOR = (1 - Pen)F_{mean}, \tag{15}$$

where $Pen = \gamma(\frac{ch}{m})^{\theta}$ and $F_{mean} = \frac{P_m R_m}{\alpha P_m + (1-\alpha)R_m}$, where γ, θ and α are parameters, ch is the number of tokens, m means the matched tokens number, P_m represents unigram precision which is computed as the ratio of the number of unigrams in the system translation that are mapped (to unigrams in the reference translation) to the total number of unigrams in the system translation, and R_m describes unigram recall which is computed as the ratio of the number of unigrams in the system translation that are mapped (to unigrams in the reference translation) to the total number of unigrams in the reference translation.

ROUGE-L takes into account sentence-level structure similarity naturally and identifies longest co-occurrence in sequence n-grams automatically. It is calculated as:

$$ROUGE - L = \frac{\sum_{S \epsilon c} \sum_g ram_l \in SCount_{match}(gram_l)}{\sum_{S \epsilon c} \sum_g ram_l \in SCount(gram_l)}, \tag{16}$$

where l means the number of tokens, $Count_{match}(gram_l)$ computes the maximum number of matched n-grams in the generated comment.

5.3 Training Details

The size of all the hidden layers of both the encoder and decoder LSTM networks are set to be 512, and the word embedding size is also set to be 512. The mini-batch size is set to be 32, 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 with 10 epochs simultaneously. We record the perplexity [37]/reward every 50 iterations. All the experiments in this paper are implemented with Python 3.6, and run on a computer with a 2.8 GHz Intel Core i7 CPU, 64 GB 1600 MHz DDR3 RAM, and a Titan X GPU with 16 GB memory, running RHEL 7.5.

Table 1: Effectiveness of code representations. (Best scores are in boldface.)

Approaches	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
TXT+HAN+DRL	19.51	2.45	0.95	0.65	5.65	31.56
AST+HAN+DRL	18.97	3.95	1.87	0.89	5.97	31.23
CFG+HAN+DRL	19.20	2.45	1.12	0.67	5.12	31.46
TXT&AST+HAN+DRL	26.56	3.96	1.89	1.32	6.21	37.68
TXT&CFG+HAN+DRL	27.66	4.25	1.97	1.12	6.38	38.24
AST&CFG+HAN+DRL	26.35	2.65	0.96	0.97	5.87	38.13
TXT&AST+AN+DRL(previous work)	25.27	10.33	**6.40**	4.41	9.29	39.13
TXT&AST&CFG+HAN+DRL	**33.16**	**12.39**	6.21	5.10	**9.43**	**46.23**
TXT&AST&CFG+HAN+DRL (with additional 20,000 subjects)	32.87	11.76	**6.32**	**5.48**	8.53	39.72

Table 2: Effectiveness of hierarchical attention network. (Best scores are in boldface.)

Attention type	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
no attention	18.76	8.21	4.98	3.46	8.24	35.28
1-layer attention (general attention)	25.79	8.45	5.73	4.67	8.79	38.49
2-layer attention	**33.16**	**12.39**	**6.21**	**5.10**	**9.43**	**46.23**

Table 3: Effectiveness of deep reinforcement learning. (Best scores are in boldface.)

Approach	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
Approach without DRL	26.89	7.21	3.76	2.31	8.21	35.78
Approach with DRL	**33.16**	**12.39**	**6.21**	**5.10**	**9.43**	**46.23**

5.4 RQ1: The Effectiveness Analysis with Different Baselines

5.4.1 Effectiveness of Code Representations

We evaluate our approach by comparing with different baseline settings (*addressing limitations 2*). Here **TXT**, **AST** and **CFG** refer to the three code representations, i.e., plain text, AST, and CFG, respectively. **HAN** refers to hierarchical attention network and **DRL** denotes deep reinforcement learning.

- **TXT+HAN+DRL.** This baseline transforms code to be plain text and uses a LSTM-based hierarchical attention network to encode the code into a hidden space, and DRL to train the model.
- **AST+HAN+DRL.** This baseline takes the sequence of the type-augmented AST as the input of the LSTM-based hierarchical attention network encoder.
- **CFG+HAN+DRL.** This baseline follows the same architecture with the above two baselines only differing in that it takes the control flow of code as the input of the LSTM-based hierarchical attention network encoder.
- **TXT&AST+HAN+DRL.** This baseline follows the same architecture as above, and it takes both the plain text and the type-augmented AST sequence of code as the input of the LSTM-based hierarchical attention network respectively. The encoded hidden vectors of them are concatenated into one hidden vector by a hybrid layer, and DRL is used to train the model.
- **TXT&CFG+HAN+DRL.** Similarly, this approach takes the plain text and the control flow of code as the input of the architecture.
- **AST&CFG+HAN+DRL.** Similarly, this approach takes the type-augmented AST sequence and the control flow of code as the input of the architecture.
- **Our previous approach [39]: TXT&AST+AN+DRL.** This approach takes the plain text and the AST as the input of the LSTM-based attention network respectively. The encoded hidden vectors of them are concatenated into one hidden vector by a hybrid layer, and DRL is used to train the model.
- **Our approach: TXT&AST&CFG+HAN+DRL.** Our proposed approach in this paper takes the plain text, the type-augmented AST sequence, and the control flow of code as the inputs of the LSTM-based hierarchical network encoder respectively. The three encoded hidden vectors are concatenated into one hidden vector by a hybrid layer. Moreover, we also apply another 20,000 subjects as our new testing dataset for an additional experiment to evaluate the robustness of our proposed approach under various projects.

Table 1 shows the experimental result comparisons between our proposed approach and the aforementioned baselines. From this table, we can observe that our proposed approach outperforms other baselines in almost all of the evaluation metrics. Compared to the baselines which use only plain text, AST or control flow, the hybrid representation of code which uses two code representations can improve the comment generation performance by 29.46% to 31.42% for the evaluation metric BLEU-1. Our proposed approach which uses three code representations outperforms the approaches which use two code representations by 16.58% to 20.53% for

BLEU-1. Compared to our previous work [39], our extended approach can improve the accuracy by about 23.79% for BLEU-1. The approaches which use two code representations outperform our previous work by 4.27% to 9.26% for BLEU-1. In addition, the testing results by our new collected data can achieve almost the same accuracy compared with the original testing dataset performance. Moreover, the results in terms of the other evaluation metrics reflect the same trends. To conclude, our approach can achieve better accuracy because of the stronger code representations and the finer-grained HAN that reflect more accurate semantic for high-quality comment generation.

5.4.2 Effectiveness of Hierarchical Attention Mechanism

To evaluate the effectiveness of hierarchical attention network (*addressing limitations 1*), we encode the code without attention network, with 1-layer attention network and with 2-layer attention network (our approach) respectively. The no attention approach encodes the input by LSTM without any attention mechanism. The 1-layer attention approach encodes the representation of the code from tokens to function directly with attention network. Our proposed 2-layer attention approach encodes the representation of the code with 2-layer attention network which considers code hierarchy—the tokens form statements and the statements construct the functions of code.

Table 2 shows the effectiveness of the hierarchical attention network. From this table, we can know that the performance of the approach with 1-layer attention network is better than the approach without any attention mechanism by 2.92% to 37.47% for different evaluation metrics. Our proposed approach (2-layer attention network) outperforms the approach with 1-layer attention network by 4.36% to 49.59% for different evaluation metrics. These results show that our proposed hierarchical attention network for code representation makes significant contributions to accurate comment generation.

5.4.3 Effectiveness of Deep Reinforcement Learning

To validate the effectiveness of the deep reinforcement learning component (*addressing limitation 3*) in our proposed approach, we train the model both with and without deep reinforcement learning component respectively, denoted as "approach with DRL" and "approach without DRL" in Table 3. From this table, we can observe that the performance of the approach with DRL outperforms the approach without DRL by 14.13% to 130.30% for different evaluation metrics. These results show that our proposed deep reinforcement learning model can significantly boost the performance of comment generation.

5.4.4 Performance Evaluation under different dataset settings

In this section, we investigate the performance of our approach under different dataset settings, i.e., cross-project and different dataset split.

To evaluate the code summarization performance of our proposed approach on cross-project dataset, we split the dataset based on their projects. In particular, we adopt 84043 pairs for training, 8689 pairs for validation, and 14390 pairs for testing from different projects. The result can be found in Table 4, from which we can observe that the results are worse compared with the randomly split dataset. To illustrate, in our approach, the generated natural language words (comments) are extracted from the collected dataset. When we conduct the cross-project experiments, it is likely that the training, validation, and testing dataset might contain different words since they are extracted from cross projects. Therefore, the generated comments are expected to be less similar with the original comments due to the discrepancy of the dictionaries among such datasets. On the other hand, it can be derived that it is essential to expand the dataset scope for improving the performance of our approach.

To investigate how our approach performs under different data split policy, we select 80% data for training, 10% data for validation, and the rest 10% for testing. The result can found in Table 4, from which we can observe that the results are close to our original data split policy. This result can validate the robustness of our approach.

5.5 RQ2: Time Consumption and Performance Trend with Different Training Epochs

We record the average training time for each epoch of this approach with different code representations as shown in Table 5. In this table, pretraining the actor network takes the first 10 epochs, pretraining the critic network takes the second 10 epochs, and training the actor-critic network takes the last 10 epochs simultaneously. From the result, we can observe that the training time of each epoch for all the three stages in our approach is less than 1 hour which is reasonable in real world.

Figure 9 shows the performance trend with the increment of the total training epoch number from 5 to 45. From this figure we can know that the performance increase from 22.98% to 34.53% in BLEU-1 with the increment of the training epochs from 5 to 45. We can also see that all the results have an approximated upward trend with the increment of the training epochs from 5 to 30, and then the performance tends to be stable. Therefore, we choose 30 as the total training epoch number in this paper.

5.6 RQ3: Performance of Different Code and Comment Length

We vary both the code and comment lengths to evaluate the effects on the representation of code and comment generation from them. Figures 10 and 11 show the performance of our proposed approach when compared with the baselines on the datasets of varying code length and comment length, respectively.

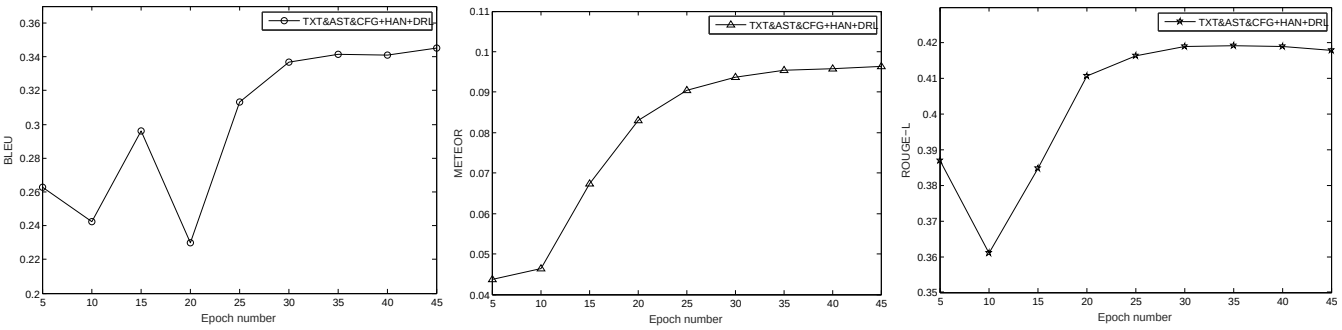
From Figure 10, we can observe that our approach performs the best when compared with other baselines on four evaluated metrics with respect to different code lengths. For BLEU-1, our approach outperforms the baselines with different code representations by 35.74%, 43.31%, 41.13%, 17.04%, 1l6.97%, and 12.66% respectively when the code length is 40. For all the evaluation metrics, the approaches which use two features for code representation, i.e., TXT&AST, TXT&CFG and AST&CFG, can always outperform the ones which use only one feature for code representation. Our approach which

Table 4: Experimental results of different dataset settings

Settings	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L
cross-project	20.10	10.54	6.33	4.36	15.87	27.35
8:1:1-split	34.12	18.02	12.33	9.18	15.34	46.32

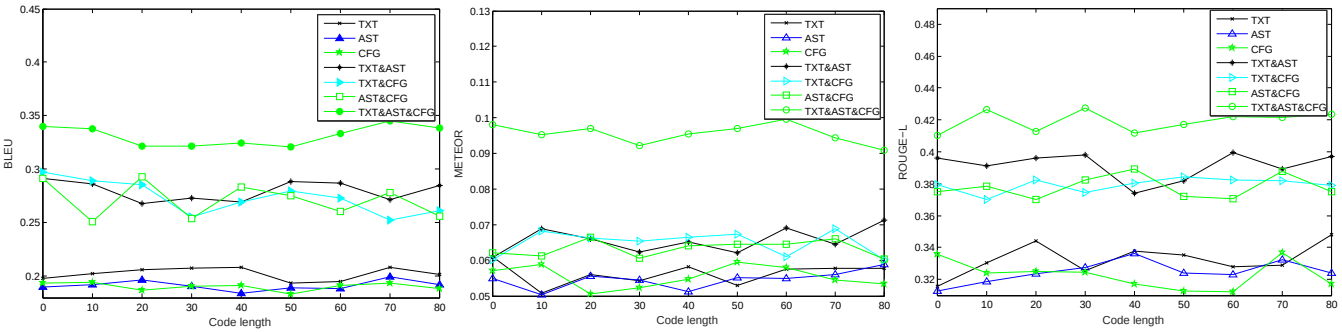
Table 5: The time consumption to train the models (mins).

	TXT	AST	CFG	TXT&AST	TXT&CFG	AST&CFG	TXT&AST&CFG
Actor pretraining epochs	20	24	23	32	31	33	39
Critic pretraining epochs	27	30	29	41	40	41	50
Actor-critic training epochs	36	41	43	50	51	53	58



(a) BLEU-1 (b) METEOR (c) ROUGE-L

Figure 9: Performance trend on different metrics w.r.t. varying training epochs.



(a) BLEU-1 (b) METEOR (c) ROUGE-L

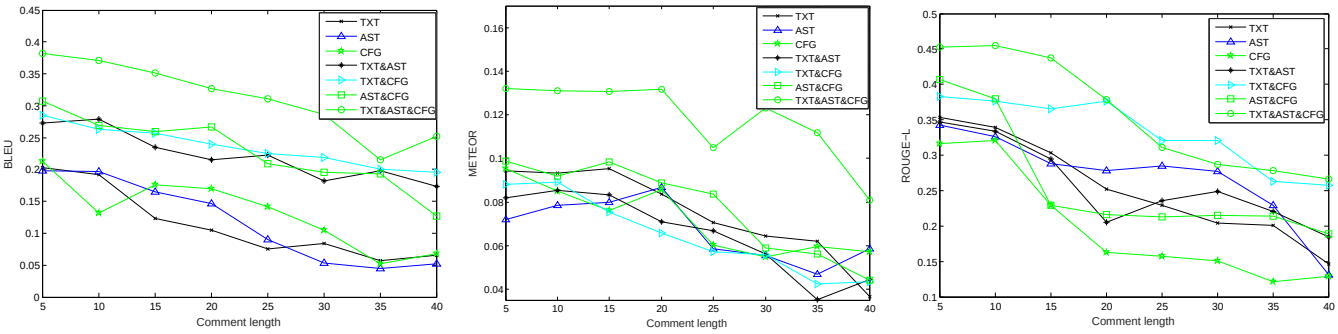
Figure 10: Performance trend on different metrics w.r.t. varying code length.

uses all the three features for code representation can outperform the ones with two features for code representation. Moreover, compared to our previous work, the extended approach can improve the comment generation accuracy by about 20.36% for BLEU-1 when the code length is 40. The results in terms of the other evaluation metrics reflect about the same trends. This further validates the effectiveness of the multi-feature code representation. Additionally, our proposed hybrid representation performs consistently under the code length ranging within (10, 80).

Figure 11 demonstrates the performance under different comment lengths. We can clearly observe that our approach has better performance compared with almost all the baselines under different comment lengths. For instance, for BLEU-1, our approach outperforms the baselines by 107.61%, 100.31%, 200.59%, 51.77%, 42.64%, and 47.77% respectively when the comment length is 20. Moreover, compared to our previous work, the extended approach can improve the comment generation accuracy by about 76.52% for BLEU-1 when the comment length is 10. From the figure we can also know that the performance becomes worse when increasing the comment length which analogizes the discoveries from the research of neural translation [78], [80].

5.7 RQ4: Performance Comparison with State-of-the-art approaches

To evaluate the code summarization performance of our proposed approach, we also select several state-of-the-art approaches, i.e., DeepCom [70], CODENN [27], Code2seq [56], and CoaCor [57] for performance comparison. In particular, DeepCom [70] utilizes the AST sequence converted by traversing the AST as the code representation



(a) BLEU-1 (b) METEOR (c) ROUGE-L

Figure 11: Performance trend on different metrics w.r.t. varying comment length.

Table 6: Code summarization results comparison with state-of-the-art approaches.

	Python				Java			
Approaches	S-BLEU	C-BLEU	METEOR	ROUGE-L	S-BLEU	C-BLEU	METEOR	ROUGE-L
DeepCom	12.92	13.27	6.09	14.33	29.65	31.55	16.87	25.76
CODENN	13.50	11.36	5.82	13.18	36.51	34.31	18.04	26.77
Code2Seq	19.49	17.53	6.52	22.04	19.96	18.07	10.17	23.56
CoaCor	25.61	23.67	9.52	29.38	33.61	32.64	17.62	28.76
Our approach	**33.16**	**30.58**	**9.43**	**46.23**	**38.25**	**36.42**	**20.01**	**37.19**

and inputs the AST sequence to the GRU-based NMT for code summarization via combining lexical and structure information. CODENN [27] uses RNN with an attention mechanism to produce comments for C# code snippets and SQL queries. Code2seq [56] represents code snippet as the set of compositional paths in its AST and uses attention to select the relevant paths while decoding. CoaCor [57] utilizes the plain text of source code and an LSTM-based encoder-decoder framework for code summarization. Apart from the dataset of Python in our previous paper, we also utilize the Java dataset applied in [70].

We evaluate the performance of all the approaches based on the aforementationed evaluation metrics (especially for BLEU, we adopt S-BLEU and C-BLEU) in terms of both Python and Java dataset. Table 6 demonstrates the code summarization results of all the approaches in terms of the selected metrics. From the results we can observe that for both the Python dataset and the Java dataset, our approach can outperform the compared approaches in terms of most evaluation metrics. Specially, in terms of the results based on the Python dataset, state-of-the-art approaches can achieve from 12.92% to 25.61% in terms of S-BLEU and from 11.36% to 23.67% in terms of C-BLEU, while our approach can achieve 33.16% and 30.58% respectively. Our approach can outperform all the compared approaches by 22.77% to 61.04% in terms of S-BLEU and by 22.60% to 62.85% in terms of C-BLEU. Such performance advantages can indicate the superiority of the our approach. Moreover, in terms of the result based on Java dataset, state-of-the-art approaches can achieve from 19.96% to 36.51% in terms of S-BLEU and from 18.07% to 34.31% in terms of C-BLEU, while our approach can achieve 38.25% and 36.42% respectively. Our approach can outperform all the compared approaches by 4.5% to 47.82% in terms of S-BLEU and by 5.79% to 50.38% in terms of C-BLEU. From the evaluation results based on both the Python and the Java projects, we can summarize that our approach can outperform multiple state-of-the-art approaches.

5.8 RQ5: Case Study and User Study

To extensively evaluate our approach, we also conduct case study and user study which are illustrated as follows.

5.8.1 Case study

We demonstrate four real-world code examples for generating their comments using our approach in Table 7. In this table, we first show the code snippet in the second line and then give the ground truth comment which is the code comment that is collected together with the code snippet from GitHub. Next, the generated comments by different approaches are given. For our approach, shown as 2-layer+DRL, we have highlighted the words that are closer to the ground truth. It can be observed that the generated comments by our approach are the closest to the ground truth. Although the approaches with DRL (1-layer+DRL) can generate some tokens which are also in the ground truth, they cannot predict those tokens which do not frequently appear in the training data, i.e., `pillar` in Case 3. On the contrary, the deep-reinforcement-learning-based approach can generate some tokens which are closer to the ground truth, such as `process`, `remove`, `subunit`. This can be illustrated by the fact that our approach has a more comprehensive exploration on the word space and optimizes the BLEU score directly.

Table 7: Case study of code summary generated by each approach.

		Case 1	**Case 2**

Code snippet		`def Pool(processes=None, initializer=None, initargs=(), maxtasksperchild=None):`
		`from multiprocessing.pool import Pool`
		`return Pool(processes, initializer, initargs, maxtasksperchild)`
			`def remove_file(source):`
			`if os.path.isdir(source):`
			`shutil.rmtree(source)`
			`elif (os.path.isfile(source) or os.path.islink(source)):`
			`os.remove(source)`

Ground truth		returns a process pool object.	remove file or directory.
Generated Comments	no attention	return a list of all available vm sizes on the cloud provider.	return a list of all available services cli example.
	1-layer	returns the total number of cpus in the system.	test the behavior of python and invalid features.
	2-layer	returns a list of all elements in a given order.	given two tp instances.
	1-layer+DRL	returns a process object with the given id.	remove a test.
	2-layer+DRL	returns a process object.	remove the file.
		Case 3	**Case 4**

Code snippet		`def ext_pillar(minion_id, pillar, url):`
		`log=logging.getLogger(__name__)`
		`data = __salt__['http.query'](url=url, decode=True, decode_type='yaml')`
		`if ('dict' code-bluein data):`
		`return data['dict']`
		`log.error(('Error caught on query to' + url) + 'More Info:'))`
		`for (k, v) in six.iteritems(data):`
		`log.error(((k + ':') + v))`
		`return {}`
			`def test_subunit_output_with_tags():`
			`state.expect=[Includes({'status': 'success', 'tags': set(['slow-ish'])}), (Includes{'status': 'success', 'tags': set(['fast-ish'])}), Includes({'status': 'success', 'tags': set()}), Includes({'status': 'success', 'tags': set()})]`
			`runner=Runner(feature_name('tagged_features'), enable_subunit=True)`
			`runner.run()`

Ground truth		read pillar data from http response.	test subunit output with tags.
Generated Comments	no attention	returns a list of all available services cli example.	return true if the given object is a valid config.
	1-layer	prints a dict of all methods on a specific server.	return a list of available audio.
	2-layer	return a string with escape-backslashes converted to nulls.	returns image.
	1-layer+DRL	test io for several.	test subunit output to unicode.
	2-layer+DRL	read pillar data.	test subunit output with unicode.

5.8.2 User study

Similar to CODENN [27], we conduct a user study to measure the output of our code summarization approach and baselines across two modalities—naturalness and informativeness. In particular, we invite 5 proficient English speakers and 5 proficient programmers with expertise in Java and Python to rate the generated comments in terms of grammaticality and fluency, on a scale between 1 and 5. First, We choose the generated comments which rank top 50 in terms of S-BLEU by our approach in both the Java and Python dataset. Furthermore, we identify their associated code snippets/original comments and their corresponding comments generated by other approaches. At last, each user randomly selects 10 out of the selected 50 comments and score them based on their respective understanding of the naturalness/informativeness of all the generated comments. The results are presented in Table 8, which demonstrates that our approach can obtain higher score compared with other baselines in both naturalness (4.35 over 3.41 to 4.18 on average) and informativeness (3.27 over 2.39 to 3.05 on average). Such results turn out to reflect the metric-based evaluation results on the fluency and consistency of our approach vs. other approaches.

6 Threats to Validity

One threat to validity is that our approach is experimented only on Python and Java code collected from GitHub, so they may not be representative of comments generation all projects using that programming language. However, as the

Table 8: Naturalness and Informativeness measures of the generated comments.

Approaches	Naturalness	Informativeness
DeepCom	3.59	2.91
CODENN	3.41	2.39
Code2seq	3.82	2.83
CoaCor	4.18	3.05
Our	4.35	3.27

components HAN and DRL in our approach are general approaches which can also be used for comment generation of other programming languages or other tasks regarding code encoding or generation.

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 [80], image captioning [81]. 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 set the BLEU score of generated sentence as the reward. It is well known that for a reinforcement learning model, 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.

7 Related Work

7.1 Deep Code Representation

With the successful development of deep learning, it has also become more and more prevalent for representing code in the domain of software engineering research. Gu et al. [82] use a sequence-to-sequence deep neural network [80], 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. [83] learn distributed vector representations using custom convolutional neural networks to represent features of code snippets, 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. [84] 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. [85] and Parisotto et al. [86] learn distributed representations of 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 [87] and other neural language models (e.g. LSTMs in Dam et al. [88]) describe context distributed representations while sequentially generating code. Ling et al. [89] and Allamanis et al. [90] combine the code-context distributed representation with distributed representations of other modalities (e.g. natural language) to synthesize code.

7.2 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 [32], statistical-language-model-based approaches [26] and deep-learning-based approaches [33], [34], [65]. Sridhara et al. [32] construct a software word usage model first, and generate comment according to the tokenized function/variable names via rules. Movshovitz-Attias et al. [26] predict comments from Java code files using topic models and n-grams. In [33], 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 code snippets into short, descriptive function name-like summaries. Iyer et al. [27] propose to use LSTM networks with attention to produce sentences that describe C# code snippets and SQL queries. In [34], the code summarization problem is modelled as a machine translation task, and some translation models such as Seq2Seq [80] and Seq2Seq with attention [91] are employed. In [92], a framework, BVAE, which includes two Variational AutoEncoders (VAEs) to model bimodal data: C-VAE for code and L-VAE for natural language is proposed. It could learn semantic vector representations for both code and description and generate completely new descriptions for arbitrary code snippets. Alon et al. [56] proposes Code2Seq, which represents code snippet as the set of compositional paths in its AST and used attention to select the relevant paths while decoding. According to the `diffs` information, Liu et al. propose NNGen (Nearest Neighbor Generator) which generate concise commit messages using the nearest neighbor algorithm [93]. CoaCor [57] utilizes the plain text of source code and an LSTM-based encoder-decoder framework for code summarization.

Unlike previous studies, we abstract more hidden information of the code for a better code representation, introduce the hierarchical attention mechanism to take the code structure into consideration and propose a deep reinforcement learning framework to accurately generate code summary.

7.3 Deep Reinforcement Learning

Reinforcement learning [49], [53], [94], 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. [54] using deep neural networks and Monte Carlo Tree Search. Human-level gaming control [95] has been achieved through deep Q-learning. A visual navigation system [96] 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 [97], dialogue generation [98] and

sentence simplification [99]. Ren et al. [97] propose an actor-critic deep reinforcement learning model with an embedding reward for image captioning. Li et al. [98] 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.

8 Conclusion

This paper presents the first hierarchical-attention-based learning approach by utilizing unstructured and structured features information of code, i.e., plain text, type-augmented AST and CFG, to reflect the hierarchical structure of code (tokens forming a statement, statements forming a function) by supporting a two-layer attention network at both token level and statement level. Our approach provides an effective representation that by differentiating tokens under different contexts for comments generation. Comprehensive experiments on a real-world dataset show that our proposed approach outperforms competitive baselines based on several standard evaluation metrics.

9 Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Shenzhen Science and Technology Program (Grant No. KQTD2016112514355531), Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. JCYJ20170817110848086) and Australia Research Council (Grant No. DP200101374, LP170100891 and DP200101328).

References

[1] B. P. Lientz and E. B. Swanson, "Software maintenance management," *IEE Proceedings E Computers and Digital Techniques Transactions on Software Engineering*, vol. 127, no. 6, 1980.

[2] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 63–72.

[3] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[4] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[5] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 579–590.

[6] J. Hua, Y. Zhang, Y. Zhang, and S. Khurshid, "Edsketch: execution-driven sketching for java," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 3, pp. 249–265, 2019.

[7] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "Smartvm: a sla-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[8] L. Zhang, "Hybrid regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 199–209.

[9] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 132–142. [Online]. Available: https://doi.org/10.1145/3238147.3238187

[10] J. M. Zhang, L. Zhang, D. Hao, M. Wang, and L. Zhang, "Do pseudo test suites lead to inflated correlation in measuring test effectiveness?" in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, 2019, pp. 252–263. [Online]. Available: https://doi.org/10.1109/ICST.2019.00033

[11] M. Wu, H. Zhou, L. Zhang, C. Liu, and Y. Zhang, "Charactering and detecting cuda program bugs," *arXiv preprint arXiv:1905.01833*, 2019.

[12] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, Korea, May 23 - 29, 2020*, 2020.

[13] I.-L. Yen, S. Zhang, F. Bastani, and Y. Zhang, "A framework for iot-based monitoring and diagnosis of manufacturing systems," in *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2017, pp. 1–8.

[14] M. Wu, Y. Ouyang, H. Zhou, L. Zhang, C. Liu, and Y. Zhang, "Simulee: Detecting cuda synchronization bugs via memory-access modeling," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, Korea, May 23 - 29, 2020*, 2020.

[15] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.*, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[16] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[17] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 268–279. [Online]. Available: https://doi.org/10.1109/ICSE.2015.47

[18] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.*, 2019, pp. 19–30. [Online]. Available: https://doi.org/10.1145/3293882.3330559

[19] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 727–738. [Online]. Available: https://doi.org/10.1145/2950290.2950295

[20] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang, "Automating cuda synchronization via program transformation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 748–759.

[21] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods review and analysis," *arXiv preprint arXiv:1709.08439*, 2002.

[22] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A programming competition benchmark for evaluating automated program repair tools," in *IEEE/ACM International Conference on Software Engineering Companion*. IEEE, 2017, pp. 180–182.

[23] E. T. Barr, M. Harman, P. Mcminn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[24] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "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, 2005, pp. 68–75.

[25] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[26] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, 2013, pp. 35–40.

[27] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[28] D. Yang, A. Hussain, and C. V. Lopes, "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, 2016, pp. 391–401.

[29] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

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

[31] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "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, 2015, pp. 574–584.

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

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

[34] T. Haije, B. O. K. Intelligentie, E. Gavves, and H. Heuer, "Automatic comment generation using a neural translation model," 2016.

[35] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[36] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

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

[38] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[39] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 397–407.

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

[41] E. Grave, A. Joulin, and N. Usunier, "Improving neural language models with a continuous cache," *arXiv preprint arXiv:1612.04426*, 2016.

[42] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 708–719.

[43] R. Rosenfeld, "Two decades of statistical language modeling: Where do we go from here?" *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1270–1278, 2000.

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

[45] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[46] S. Thrun and M. L. Littman, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 285–286, 2005.

[47] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

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

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

[50] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[51] Y. Keneshloo, T. Shi, C. K. Reddy, and N. Ramakrishnan, "Deep reinforcement learning for sequence to sequence models," *arXiv preprint arXiv:1805.09461*, 2018.

[52] V. Konda, "Actor-critic algorithms," *Siam Journal on Control & Optimization*, vol. 42, no. 4, pp. 1143–1166, 2003.

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

[54] 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, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[55] "Genism," https://pypi.org/project/gensim/.

[56] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[57] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: Code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference*. ACM, 2019, pp. 2203–2214.

[58] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison Wesley*, vol. 7, no. 8, p. 9, 1986.

[59] "Abstract syntax trees," accessed 16 August 2018. https://docs.python.org/2/library/ast.html.

[60] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, L. Yang, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[61] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL, 2016, pp. 1480–1489.

[62] D. Hu, "An introductory survey on attention mechanisms in nlp problems," *arXiv preprint arXiv:1811.05544*, 2018.

[63] S. Jain and B. C. Wallace, "Attention is not explanation," *arXiv preprint arXiv:1902.10186*, 2017.

[64] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[65] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[66] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu, "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, 2002, pp. 311–318.

[67] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[68] "Github," https://github.com/.

[69] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.

[70] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, pp. 1–39, 2019.

[71] S. Banerjee and A. Lavie, "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, 2005, pp. 65–72.

[72] C. Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[73] E. M. Ponti, R. Reichart, A. Korhonen, and I. Vulic, "Isomorphic transfer of syntactic structures in cross-lingual nlp," in *The 56th Annual Meeting of the Association for Computational Linguistics*, 2018, pp. 1531–1542.

[74] T. A. Pirinen, "Neural and rule-based finish nlp models-expectation, experiments and experiences," in *The fifth Workshop on Computational Linguistics for Uralic Languages*, 2019, pp. 104–1114.

[75] H. Shi, H. Zhou, J. Chen, and L. Li, "On tree-based neural sentence modeling," *arXiv preprint arXiv:1808.09644*, 2018.

[76] K. Nguyen, H. D. Iii, and J. Boyd-Graber, "Reinforcement learning for bandit neural machine translation with simulated human feedback," in *Conference on Empirical Methods in Natural Language Processing*, 2017.

[77] J. Pouget-Abadie, D. Bahdanau, B. V. Merrienboer, K. Cho, and Y. Bengio, "Overcoming the curse of sentence length for neural machine translation using automatic segmentation," *arXiv preprint arXiv:1409.1257*, 2014.

[78] J. Li, D. Xiong, Z. Tu, M. Zhu, Z. Min, and G. Zhou, "Modeling source syntax for neural machine translation," *arXiv preprint arXiv:1705.01020*, 2017.

[79] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. V. D. Oord, and K. Kavukcuoglu, "Neural machine translation in linear time," *arXiv preprint arXiv:1610.10099*, 2016.

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

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

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

[83] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016, pp. 1287–1293.

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

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

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

[87] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, 2014, pp. 649–657.

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

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

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

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

[92] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 826–831.

[93] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 373–384.

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

[95] 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.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

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

[97] Z. Ren, X. Wang, N. Zhang, X. Lv, and L. J. Li, "Deep reinforcement learning-based image captioning with embedding reward," in *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE, 2017, pp. 1151–1159.

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

[99] X. Zhang and M. Lapata, "Sentence simplification with deep reinforcement learning," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 584–594.