Adaptive Prior-Knowledge-Assisted Function Naming Based on Multi-level Information Explorer

Lancong Liu, Shizhan Chen, Sen Chen, Guodong Fan, Zhiyong Feng, Hongyue Wu *
College of Intelligence and Computing, Tianjin University, Tianjin, China
Email: (llancong, shizhan, senchen, guodongfan, zyfeng, hongyue.wu)@tju.edu.cn

***Abstract*—Automatic function naming aims to generate a concise and meaningful name for a function, and has become a popular research area. Function naming models based on deep-learning have made significant progress in recent years. Most of the existing neural models represent a function based on the granularity of token or AST (Abstract Syntax Tree) node. However, generating function names requires more fine-grained knowledge of code, but the representation of tokens or AST nodes is not enough to capture global function semantics. In our work, we propose Apker, a novel Adaptive prior-knowledge-assisted function naming based on multi-level information explorer. The Apker includes three modules: Multi-level Information Explorer (MIE), Adaptive Prior Knowledge Adaptor (APKA) and the Generator. The MIE captures the function semantics from a local and global perspective, motivated by the understanding patterns of humans, who will first understand the meaning of each statement and then comb their logical relations to understand the whole function. The APKA uses the pre-retrieved prior knowledge to assist the model, motivated by our observation that *certain name tokens can be extracted directly from certain statements and such probability differs significantly in different types of statements*. Finally, the Generator generates function names. The experimental results demonstrate that our approach outperforms the baselines by 5.4% in Precision, 12.7% in Recall, and 7.4% in F1-score.**

***Index Terms*—function name generation, code summarization**

I. Introduction

A study has shown that developers spend more time in program comprehension than coding [13]. Developers need to provide an understandable function name because such names can help them to understand a function quickly without reading the body in detail for further information. What's more, inappropriate names may lead to software defects. For example, Abebe et al. [1] find that inconsistent identifier use contributes to the faultiness of classes. For novelties, it is difficult to generate a function name that can directly reflect their intent. Therefore, automatically generating function names becomes a critical task in reverse engineering, which can suggest appropriate names for developers.

Most existing methods use data-driven models to mine potential information from source code and then convert it into forms that can be understood easily by neural networks, e.g., some studies construct code representations from split tokens [2] and parsed AST [3, 6, 7]. Despite their effectiveness, such approaches are based on the granularity of token or AST nodes. Specifically, they input tokens or AST nodes to an encoder to learn their representations and then decode them to generate a function name. Representing source code in such way is not enough to capture global function semantics.

DOI reference number: 10.18293/SEKE2022-085
*Corresponding Author

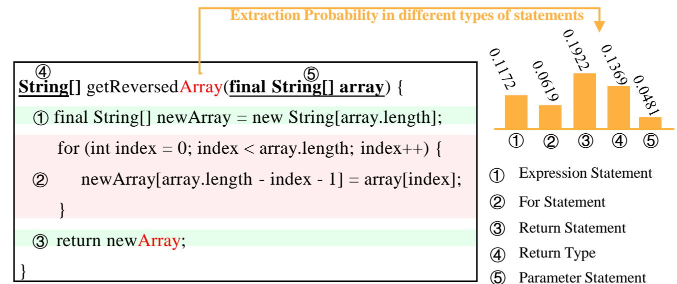

Fig. 1. Example to illustrate our motivations. We decompose the "getReversedArray" function into five statements. Each statement has its own type and we tag them respectively. The extraction probability is the prior knowledge used in the Apker.

To solve this limitation, we propose a novel approach, named Apker. There are two motivations behind the Apker. First, to mine more fine-grained knowledge of code, we follow the humans' way of understanding a function. Given a function, humans will *try to understand the meaning of each statement; then comb their logical relations to understand the whole function* [5]. Therefore, capturing the function semantics based on statement-level may be more efficient than token or AST node. Second, we observe that *certain name tokens can be extracted directly from certain statements and such probability differs significantly in different types of statements*. Such observations can be a branch to assist the model to generate name tokens correctly. Take Figure 1 as an example, there are three name tokens: "get", "reversed" and "array", among which the first two tokens don't appear in any statements, thus, they couldn't be extracted from any of them. For these tokens, the model must predict them according to its captured semantics. However, the token "array" can be extracted from the statements and it is most likely to be extracted from the "return statement" according to the prior knowledge.

To model the above working patterns, Apker introduces three modules, i.e., **M**ulti-level **I**nformation **E**xplorer (MIE), **A**daptive **P**rior **K**nowledge **A**daptor (APKA) and the Generator. The MIE captures each statement semantics from a local

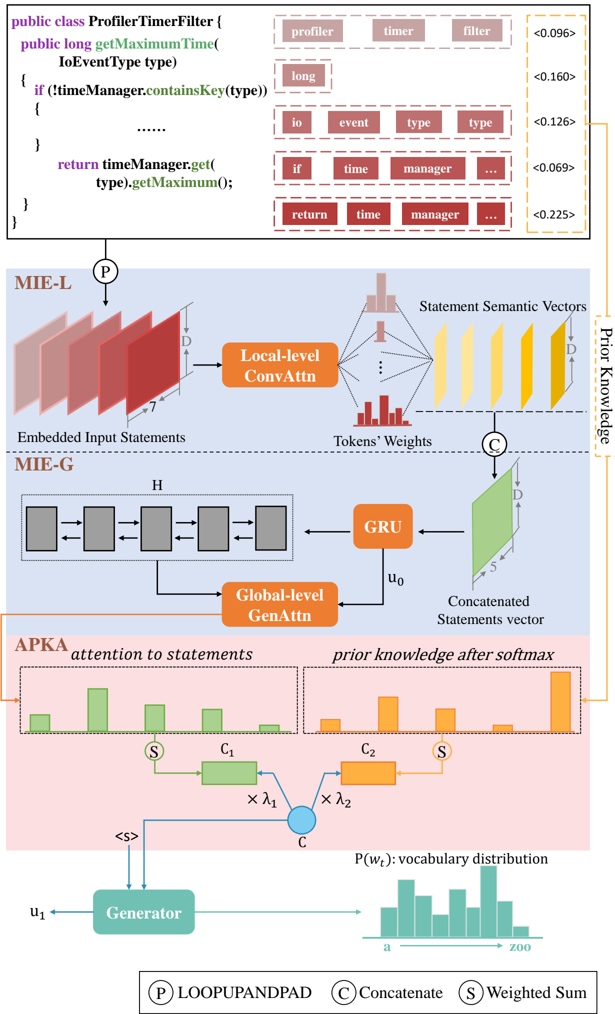

Fig. 2. Illustration of our proposed approach. This only shows the procedure of generating function names at the first time step.

perspective, based on which, it captures the whole function semantics from a global perspective. The APKA uses the prior knowledge to assist the model. Here, the prior knowledge means *The extraction probability for certain name tokens differs significantly in different types of statements*. Finally, the Generator decodes function names auto-regressively.

In summary, our main contributions are as follows:

- To explore global information by aggregating more fine-grained local information, we design the multi-level information explorer for the task of function naming.
- We fuse the prior knowledge in an adaptive way to assist the model to learn.
- The experiments and analysis on the public dataset verify the effectiveness of our approach, which is able to outperform previous state-of-the-art approaches.

II. Related Work

In this section, we introduce the related works according to the granularity of representing a function.

Early studies regard source code as plain text and employ split tokens to learn code semantics [2, 12, 14, 17, 23]. A brief history of relevant works starts with [12]. In their work, they aim to automatically generate summaries from code snippets collected from StackOverflow. They split a code snippet into tokens and apply neural machine translation networks with attention [21] to generate code summaries. Later, Allamanis et al. [2] designs a novel attention network purely based on convolutional blocks for extreme code summarization and evaluates that their network performs better than general attention.

Code is more structural than plain text. Therefore, researchers devote themselves to parsing code structure by using code analysis tools. Parsing source code into an AST is leveraged by various approaches as AST can obtain the syntax structure information [4, 5, 6, 10, 11, 15, 16, 18, 22]. At the first stage, researchers convert the ASTs into sequences before they are fed into the model. For example, deep learning similarity [22] expresses its parsed AST as a stream of the nodes by performing a pre-order visit of the sub-tree. However, converting AST to a flattened sequence destroys the original structural information. Thus, later works focus on finding a way of representing code structure directly. For example, Shido et al. [18] designs Tree-LSTM as the extension of LSTM. Tree-LSTM can learn tree structures in ASTs directly by propagating information from leaves to the root and is more effective than a sequential model used for machine translation in natural language process (NLP) when applied to source code summarization. Besides AST, other information such as data-flow and control-flow is captured [3, 7, 8].

Despite their effectiveness, all these approaches represent the source code based on the granularity of token or AST node, which is limited to capture global semantic information. What's more, they obey humans' reading habits when understanding a function.

III. The Overall Architecture of Apker

In this section, we first give a problem definition, and then introduce the three modules separately.

A. Problem Definition

The goal of this study is to create a model able to generate a function name in Java source code. As shown in Figure 2, given a function $F = \{S_0, S_1, \cdots, S_{n-1}\}$, where S_i denotes a statement, e.g., we decompose the function into five types of statements and the number behind them is the pre-counted prior knowledge, the Apker receives embedded statements as input and outputs name tokens step by step. All name tokens compose the final function name $N = \{n_0, n_1, \cdots, n_{m-1}\}$, where m is the number of tokens composing the predicted function name.

The prior knowledge of S_i is a probability and it is calculated as the number of tokens that compose the target function divided by the total number of tokens under such a statement. For example, there are 33,128,737 tokens under statements with "ClassName" type in total, among which

5,359,581 tokens exist in target function name. Therefore, the prior knowledge of statements with "ClassName" type is 0.1618. As Coganc [23] has counted the probability of 33 types of statements, we directly use it as our prior knowledge.

B. Multi-level Information Explorer (MIE)

In this paper, we introduce MIE, which includes a local-level part (MIE-L) and a global-level part (MIE-G).

MIE-L MIE-L is to capture each input statement semantics from a local perspective. As different tokens have different influences on a statement, we first use a Convolutional Attention Network (ConvAttn) [2] to weight tokens, then calculate their weighted sum and use it as the statement semantics vector. The detailed implementation is shown in Algorithm 1.

Algorithm 1 Generate a statement semantics vector

1: $ConvAttn(Embedded\ Statement\ E_{S_i})$
2: $L_1 \leftarrow RELU(CONV1D(E_{S_i}, K_{l_1})$
3: $L_2 \leftarrow CONV1D(L_1, K_{l_2})$
4: $L_{feat} \leftarrow L_2/||L_2||_2$
5: $L^i_{weight} \leftarrow SOFTMAX(CONV1D(L_{feat}, K))$
6: $\overset{\wedge}{n_i} \leftarrow \sum_{j=0}^{T-1} (L^i_{weight})_j (E_{S_i})_j$
7: $return\ \overset{\wedge}{n_i}$

Here, $L^i_{weight} = \{L^i_{weight_0}, L^i_{weight_1}, \cdots, L^i_{weight_{T-1}}\}$, where $L^i_{weight_j}$ is the jth token weight under the ith statement, and $\overset{\wedge}{n_i}$ is our obtained statement vector.

MIE-G MIE-G is to learn correlations between obtained statement vectors $\overset{\wedge}{n_i}$s and then score each statement according to its influence on the whole function semantics, which will be used for generating the final function vector. We use GRU to learn correlations in Equation (1) and employ the standard attention mechanism (GenAttn) to score statements in Equation (2).

$$\begin{aligned} CL &= Concat(\overset{\wedge}{n_0}, \overset{\wedge}{n_1} \cdots, \overset{\wedge}{n_{n-1}}) \\ H, u_t &= GRU(CL) \end{aligned} \tag{1}$$

where CL is the concatenated statement vectors and $H = \{h_0, h_1 \cdots, h_{n-1}\}$.

$$\alpha_{it} = \frac{exp(e(h_i, u_{t-1}))}{\sum_{i=0}^{n-1} exp(e(h_i, u_{t-1}))} \tag{2}$$

Here, $e(h_i, u_{t-1}) = a(u_{t-1}, h_i)$, a is a feedforward neural network (FNN). α_{it} denotes scores of statement S_i at time step t.

C. Adaptive Prior Knowledge Adaptor (APKA)

In fact, most of the words composing a function name can be extracted from the statements directly. Thus, when predicting a name token, if there is a thing that can assist the model to locate the statement, it will generate a correct result more quickly and correctly. This is just the functionality of the prior knowledge.

In APKA, we fuse the prior knowledge and the model in an adaptive way as shown in Equation (3). The reason we design in such way is that the model should judge if this name token can be extracted or predicted at each time step.

$$\begin{aligned} C_1 &= \sum_{i=0}^{n-1} \alpha_{it} h_i \\ P &= SOFTMAX(p_0, p_1, \cdots, p_{n-1}) \\ C_2 &= \sum_{i=0}^{n-1} P h_i \\ \lambda_1, \lambda_2 &= \sigma(u_t W_u + C_1 W_{C_1} + C_2 W_{C_2}) \\ C &= \lambda_1 C_1 + \lambda_2 C_2 \end{aligned} \tag{3}$$

Here, C_1 and C_2 are context vectors generated by statement attentions and the prior knowledge respectively. p_i is the prior knowledge of S_i. W_u, W_{C_1} and W_{C_2} are learnable parameters. The computed λ_1, $\lambda_2 \in [0, 1]$ weight the expected importance of C_1 and C_2 respectively and their values can show the adaptability.

D. Generator

Generator aims to decode the final function vector C into an actual function name. For each step t, we predict the tth word by generating the vocabulary distribution. Generator takes the current input word $x_t (x_0 =< s >)$, last hidden state u_{t-1} and C as input. We describe it with Algorithm 2.

Algorithm 2 Generate vocabulary distribution of each step t

1: $Generator(x_t, u_{t-1}, C)$
2: $e_{x_t} \leftarrow Embed(x_t)$
3: $input \leftarrow Concat(e_{x_t}, C)$
4: $output, s_t \leftarrow GRU(input, u_{t-1})$
5: $output \leftarrow Linear(Concat(output, C))$
6: $P_{w_t} \leftarrow SOFTMAX(output)$
7: $return\ P_{w_t}, u_t$

During training, the overall loss for the whole sequence is calculated as the average loss at each time step shown in Equation (4), which is the negative log likelihood of the word w_t for that step:

$$loss = \frac{1}{T} \sum_{t=0}^{T} (-logP(w_t)) \tag{4}$$

IV. Experiments

In this section, we firstly describe used dataset as well as some widely-used metrics and experimental settings in detail. Then we present the evaluation and analysis of the proposed approach.

A. Datasets, Metrics and Settings

Dataset We use the Java dataset collected by [2], which is obtained from 11 open-source Java projects on GitHub. This dataset has been split into training/testing/validation by projects. Particularly, we remove the functions whose length of their names is less than 2 or longer than 6 because

TABLE I
COMPARISONS OF OUR APPROACH AND THE EXISTING METHODS

Approaches	Metrics			Used Code Form					
	Precison↑	Recall↑	F1↑	tokens	AST	DE-AST	CFG	AST path	stmts
Cognac	0.671	0.597	0.632	√					
ConvNet	0.459	0.394	0.406	√					
TBCNN	0.409	0.318	0.355		√				
TreeCaps	0.526	0.414	0.468		√				
GGNN	0.403	0.353	0.369			√			
GREAT	0.473	0.400	0.436	√		√			
Sequence GINN	0.648	0.562	0.602				√		
Code2vec	0.234	0.220	0.214					√	
Code2seq	0.504	0.354	0.426	√				√	
Apker	**0.701**	**0.673**	**0.679**	√					√

TABLE II
EFFECTIVENESS OF MIE.

Attention type	ConvAttn	GenAttn	APKA	Metrics		
				Precision	**Recall**	**F1**
Basic				0.469	0.411	0.431
+ConvAttn	√			0.609	0.556	0.573
+GenAttn		√		0.553	0.504	0.520
MIE	√	√		**0.666**	**0.628**	**0.638**

such functions are not practical. Our dataset contains 163,168 functions finally.

Metrics We measure prediction performance using Precision, Recall, and F1 over the sub-words in generated names, following the metrics used by [5, 6, 7].

Settings To train our model, we optimize the objective using stochastic gradient descent with RMSProp [20] and Nesterov momentum [9]. We use dropout [19] on all parameters and gradient clipping. Each of the parameters in the model is initialized with normal random noise around zero, except for W_u, W_{C_1}, and W_{C_2} in APKA are initialized with kaiming normal. For ConvAttn, the best values for k_1, k_2, w_1, w_2, and w_3 are 8, 8, 24, and 29. The embed and hidden dimensions are set to 100. The dropout rate is set to 25%. The batch size is set to 32.

B. Comparison with State-of-the-art Methods

We compare our approach with a wide range of state-of-the-art function naming models, i.e., Cognac [23], ConvNet [2], TBCNN [16], TreeCaps [6], GGNN [3], GREAT [8], Sequence GINN [24], Code2Vec [5] and Code2Seq [4]. As shown in Table I, our Apker outperforms state-of-the-art approaches across all metrics. Compared to these best scores of different metrics, our method has a Precision improvement of 5.4%, a Recall improvement of 12.7% and an F1-score improvement of 7.4%. The improved performance demonstrates the validity of our approach.

C. Effect of MIE

ConvAttn and GenAttn are the two key components to implement the MIE module. Therefore, to evaluate the effectiveness of MIE, we make an ablation study.

Here, we construct four baseline networks. Note that we all remove the APKA module in these four networks. The first one (denoted as "Basic") is to remove the two attention mechanisms. The second one ("+ConvAttn") employs the original MIE-L to weight token attentions for each statement, while the third one ("+GenAttn") uses the same MIE-L as the first one but adds GenAttn after getting final statement vectors. The last baseline network ("MIE") is our approach without APKA.

Table II shows the effectiveness of MIE. From this table, we can know that both "+ConvAttn" and "+GenAttn" outperform "Basic" on all metrics and "+ConvAttn" has a significant improvement compared to "+GenAttn". It indicates that the generated statement vector considering different weights of tokens reflects the statement semantics better. Moreover, MIE outperforms "+ConvAttn" for all different evaluation metrics. These results show that our proposed MIE makes significant contributions to accurate function name generation.

TABLE III
EFFECTIVENESS OF APKA.

Measure	MIE	+SPKA	+APKA
Precision↑	0.666	0.682	**0.701**
Recall↑	0.628	0.659	**0.673**
F1-score↑	0.638	0.663	**0.679**

D. Effect of APKA

In order to evaluate the performance of APKA, we implement a static prior knowledge adaptor named SPKA to make comparisons in Equation (5). The difference between SPKA and APKA lies in the way to fuse the prior knowledge.

$$\alpha_{it} = \alpha_{it} + P$$
$$C = \sum_{i=0}^{n-1} \alpha_{it} h_i \quad (5)$$

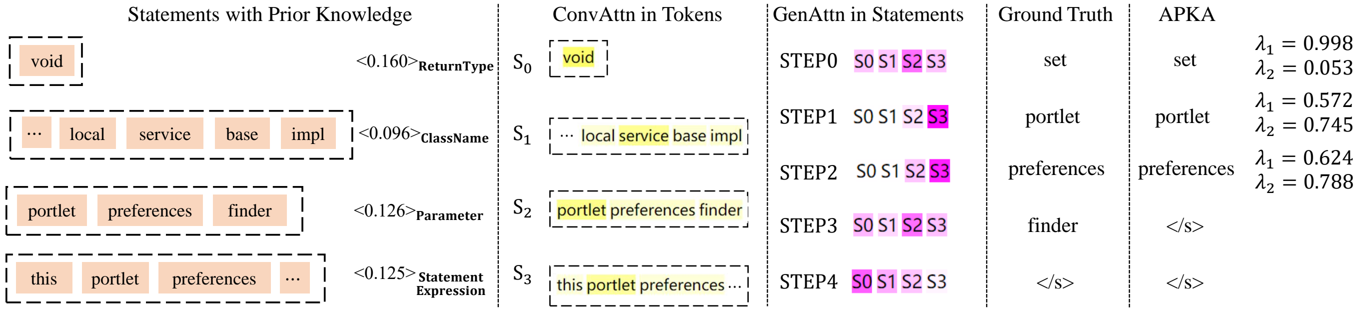

Fig. 3. We give the visualization of our approach. In the *Statements with Prior knowledge* step, each dashed box refers to a statement and it has the prior knowledge behind. S_i refers to the ith statement. Tokens highlighted in yellow and statements highlighted in purple represent their attention scores. All colors follow a principle: the darker a color is, the more important an item is. Each pair of λ_1 and λ_2 denotes the weights of the model and the prior knowledge at each time step respectively.

As shown in Table III, it is clear that our APKA successfully boosts the performance, verifying its effectiveness. Adding the prior knowledge to the statement attentions directly means the totally same extraction probability for all time steps, ignoring the dynamic probability to extract or to predict at each time step.

E. Visual Analysis

In Figure 3, we give an illustrative example to better understand our approach. As we can see, in Apker, the ConvAttn first learns statement semantics by scoring their inner tokens, e.g., the first statement S_0 only has a token "void", thus ConvAttn scores it high; S_1 attends more to the token "service" and the last two statements both attend more to the token "portlet" than others. Then the GenAttn scores these statements according to their influence on the whole function semantics.

Now, look at how APKA takes effect. As the name token "set" doesn't appear in any statement, the APKA is expected to weigh more in the model rather than the prior knowledge. As we can see, the λ_1 and λ_2 are 0.998 and 0.053 respectively, satisfying our expectations. But When generating "portlet" and "preferences", the APKA should attend to the prior knowledge because the two tokens can be extracted from the last two statements directly. Their λ_1s and λ_2s are also corresponding to our expectations. Though the last name token "finder" is not predicted rightly, the presented result can still verify the capability of Apker to generate function names.

V. Discussion

In this section, we discuss the strength and some threats of Apker.

A. Strength of Apker

We have evaluated three main advantages of Apker that may explain its effectiveness in function naming: (a) *A more comprehensive representation of source code.* The Apker decomposes a function into several types of statements. The experiments also validate the capability of statement-level approach than token or AST nodes. (b) *A multi-level attention mechanism.* The Apker uses a multi-level attention mechanism, among which, local-level attention infers the contribution of each token to the statement and global-level infers the contribution of each statement to the whole function.

B. Threats to Validity and Limitations

Our proposed Apker may suffer from two threats. One threat is on the prior knowledge. The obtained statement prior knowledge is highly dependent on the empirical study. The correctness of the prior knowledge has an influence on the predicted results.

Another threat lies in the extensibility of Apker. Our model needs to identify different types of statements based on a static analysis tool. There are many tools to analyze Java source code but few in other programming languages. Therefore, it may be difficult to extend our approach to other languages. Besides, we also consider the class name of the target function and other statements in caller/callee functions. However, they can only be parsed from a whole program. Therefore, it is challenging to extend Apker to some datasets collected from other channels where only a small code snippet can be extracted.

VI. Conclusion

In this paper, we proposed a novel prior-knowledge-guided neural network for the task of function naming. It includes a multi-level information explorer to capture local information with learning to aggregate them to global information and an adaptive prior knowledge adaptor. Two attention mechanisms (ConvAttn and GenAttn) were used so that different contributions of tokens to a statement and statements to a function can be inferred. In particular, we fuse prior knowledge in Apker to assist the model. We demonstrate the superior performance of the proposed framework. For future work, we plan to look into more accurate and valuable prior knowledge. Furthermore, we will conduct comprehensive experiments on other programming languages such as Python and C.

VII. Acknowledgement

This work is supported by the National Natural Science Key Foundation of China grant No.61832014 and No.62032016, National Natural Science Foundation of China grant No.

62102281, and the Natural Science Foundation of Tianjin City grant No.19JCQNJC00200.

References

[1] Surafel Lemma Abebe, Venera Arnaoudova, Paolo Tonella, Giuliano Antoniol, and Yann-Gael Gueheneuc. Can lexicon bad smells improve fault prediction? In *2012 19th Working Conference on Reverse Engineering*, pages 235–244. IEEE, 2012.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[6] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 30–38, 2021.

[7] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

[8] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[9] Geoffrey Hinton, Nitsh Srivastava, and Kevin Swersky. Neural networks for machine learning. *Coursera, video lectures*, 264(1):2146–2153, 2012.

[10] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[11] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020.

[12] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke 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)*, pages 2073–2083, 2016.

[13] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

[14] Yi Li, Shaohua Wang, and Tien N Nguyen. A context-based automated approach for method name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 574–586. IEEE, 2021.

[15] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[16] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

[17] Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1372–1384, 2020.

[18] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[20] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[22] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553. IEEE, 2018.

[23] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 741–753, 2021.

[24] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.