Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach*

Rui Xie[1,2], Wei Ye[1†], Jinan Sun[1], and Shikun Zhang[1]
[1] National Engineering Research Center for Software Engineering, Peking University, Beijing, China
[2]School of Software and Microelectronics, Peking University, Beijing, China
ruixie, wye, sjn, zhangsk@pku.edu.cn

***Abstract*—Code summaries are brief natural language descriptions of source code pieces. The main purpose of code summarization is to assist developers in understanding code and to reduce documentation workload. In this paper, we design a novel multi-task learning (MTL) approach for code summarization through mining the relationship between method code summaries and method names. More specifically, since a method's name can be considered as a shorter version of its code summary, we first introduce the tasks of generation and informativeness prediction of method names as two auxiliary training objectives for code summarization. A novel two-pass deliberation mechanism is then incorporated into our MTL architecture to generate more consistent intermediate states fed into a summary decoder, especially when informative method names do not exist. To evaluate our deliberation MTL approach, we carried out a large-scale experiment on two existing datasets for Java and Python. The experiment results show that our technique can be easily applied to many state-of-the-art neural models for code summarization and improve their performance. Meanwhile, our approach shows significant superiority when generating summaries for methods with non-informative names.**

***Index Terms*—code summarization, method name prediction, multi-task learning, deliberation network**

I. Introduction

Source code summarization is the task of creating readable summaries that describe the functionality of source code pieces [2]. High-quality code summarization not only greatly frees programmers from their tedious work on code documentation [3], but also benefits source code maintenance and improves the performance of code retrieval [4]. The task of code summarization mainly involves two different but related scenarios. The first scenario is to predict a natural language sentence that describes a given method. Researchers have built corresponding datasets by extracting source code of individual methods and the first sentence of their documentation (e.g., docstring of Python method [4], [5] and Javadoc of Java method [1]–[3], [6]). The second scenario is to predict a natural language sentence that describes a given code snippet [7], [8], e.g., from Stack Overflow, which is not necessarily the source code of an entire method. We focus on the first scenario in this paper.

Deep learning is now widely used in code summarization as a mainstream approach. More and more techniques of neural network for machine translation and text summarization have been introduced into code summarization recently ever since Iyer et al. [9] proposed a classical neural attention model. Researchers in the fields of natural language processing, deep learning, and software engineering have proposed a variety of neural models, which mainly involve advances in two following directions. The first direction is to design better feature representation to capture more accurate semantics of code such as control-flow structure information of a method [3], [7]. The second direction is to bring in more sophisticated deep learning techniques of text generation, such as variational auto encoder [10], dual learning [11], [12], reinforcement learning [4], et al. Although earlier studies achieved considerable performance improvement, the relationship between method names and code summaries, which could improve code summarization significantly as we demonstrated in this work, has not been well explored yet.

As shown in Table I, although the name and code summary of a method have a large difference in length, they both can be considered as the functional description of the method code. We analyzed the method name and code summary in the dataset built by Leclair et al. [13], and found that averagely 50.6% of the words in method names appear in the corresponding summaries, and 21.3% of the words in summaries appear in the corresponding method names. For about 20% of the methods, all the words in the method names appear in the corresponding summaries. In fact, programmers are also do some kinds of code summarization work in the process of naming a method, and they usually write code summaries based on the corresponding method name. Therefore, the task of Method Name Generation (MNG), which predicts a method name with its given body [7], [14], [15], can be used as an auxiliary task of code summarization to provide a useful inductive bias. Adding this auxiliary task will make the model more focused on those hypotheses that can explain both code summarization and MNG at the same time, and consequently have a potential to improve the generalization and performance

† Corresponding author.
* This research is supported by the 2019 Industrial Internet Innovation and Development Project - Security Detection Tool on Source Code of Industrial Software, No. TC190H46G/1.

Human-Written Summary: return the given decimal number formatted for a specific locale.

Source Code:

```
def format_decimal(number, format, locale):
    locale = Locale.parse(locale)
    if (not format):
        format = locale.decimal_formats.get(format)
    pattern = parse_pattern(format)
    return pattern.apply(number, locale)
```

Method Name	Generated summary
format_decimal	return the decimal number formatted for locale.
f_dec	create a regex which doesnt like pattern but set it to.
formatdecimal	set the locale to a pattern.
formt_decimal	set the locale to a pattern.

Fig. 1. Generated Summaries generated by a state-of-the-art model Ast-attendgru [1] with different method names. The model generates a high-quality summary when we feed an expected informative method name (*format_decimal*) into the model. We then replace the method name with ones which involve customized abbreviations(*f_dec*), nonstandard conjunction(*formatdecimal*), and a spelling error(*formt_decimal*) to simulate the scenario of summarizing the poorly-named methods. The quality of generated summaries is severely degraded.

TABLE I
EXAMPLES OF METHOD NAME AND CODE SUMMARY.

Method Name	Code Summary
update state	called when a command update its state
publish	publish a tt log record tt
tool enabled	invoked if a tool was enabled
get type	get the session manager event type

of the model.

However, poorly-defined method names, which are common in practical software projects [16], [17], could lead to inaccurate code summaries. As shown in Figure 1, method names which contain customized abbreviations, nonstandard conjunction, or spelling errors may mislead the code summarization model. In other words, the state-of-the-art models are highly dependent on high-quality method names. In this case, if MNG could first produce a more meaningful method name, the summarization model can make further extension and polishing based on a more informative compressed draft (i.e., the generated method name). Inspired by the high-level concept of "deliberation" proposed by Xia et al. [18], we incorporate another auxiliary task of method name informativeness prediction (MNIP) into a novel two-pass deliberation process in our MTL architecture, aiming to generate consistent code summaries more robustly, especially when method names are non-informative.

Last but not least, since the method name is always a part of the method code, the MNG task has a natural large-scale self-labeled training corpus, which resembles the language model in natural language processing. Language models predict the next (or masked) word in a given text sequence, which can be trained and used to improve almost all other NLP tasks, thanks to extremely large self-supervised corpora [19]–[21]. In a similar way, we show that by pre-training MNG, the self-supervised task, with an extra large-scale corpus, we could further improve code summarization.

In short, by introducing tasks MNG and MNIP, we design a deliberation multi-task learning approach, which better characterizes the relationship between method summaries and method names, to improve code summarization. To evaluate our approach, we carried out large-scale experiments on two existing datasets for Java and Python. We also collected a huge self-labeled dataset (30.4 million pairs of method bodies and their names) from Github for evaluating the pre-training effect of task MNG. From the experiment results, we found that exploiting the inner connection between code summaries and method names can significantly improve the performance of code summarization. We believe the deliberation MTL architecture has the potential to be extended to more similar scenarios, e.g., mining inner connections of summaries and headlines of internet news.

Our main contributions are:

- We proposed a Deliberation Multi-task learning Approach for COde Summarization (DMACOS). To the best of our knowledge, we are the first to explore the inner connection between method names and code summaries systematically in the research of code summarization.
- We carried out an experiment on two large existing datasets for code summarization (on Java and Python). The experiment results show that our MTL approach can not only achieve competitive performance, but also be easily applied to some existing encoder-decoder models of code summarization and improve their performance.
- We studied DMACOS's performance on methods with non-informative names, and DMACOS showed significant superiority brought by its two-pass deliberation mechanism.
- We showed extra knowledge introduced by MNG model pre-trained with large-scale self-supervised corpus can further improve the performance of the model, which can be observed even across programming languages.

II. PROPOSED APPROACH

A. Overall Architecture

In this section, we first present an architecture overview of our deliberation multi-task learning approach. The three tasks

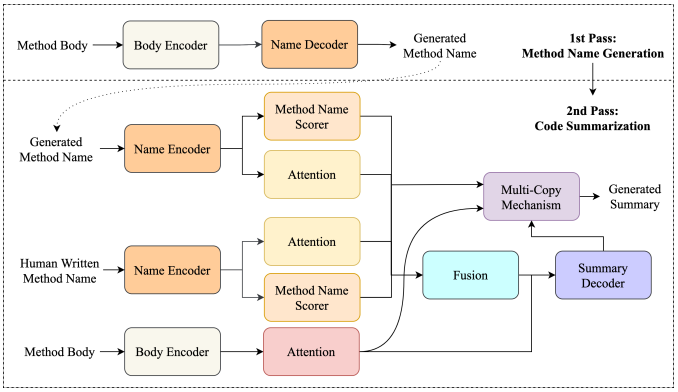

Fig. 2. The overall architecture of DMACOS. In the first pass, we generate the method name (a compressed version of summary). Then the 2nd pass incorporates a deliberation process to generate the final code summary, by feeding and fusing the output of the first-pass into the summary decoder. The links from method body encoder to informativeness predictor on the second pass are omitted for readability.

supported by our architecture are as follows:

- **The task of code summarization (COS)**, which generates a natural language description for the given method, is the target task in our approach.
- **The task of method name generation (MNG)**, which predicts a method name with the given the method source code, is the auxiliary task for code summarization.
- **The task of method name informativeness prediction (MNIP)**, which predicts a real number between 0 and 1 indicating how informative the method name is. We use the proportion of words in a method name appearing in the corresponding code summary as the golden informativeness score.

Like most tasks of text generation, COS and MNG can be both generally formulated as encoder-decoder models. We have a method name encoder and a method body encoder in DMACOS. The three tasks share the method body encoder, and COS and MNIP share the method name encoder. Ihe input of the method body encoder is method code with the method name masked by a special token "<name>".

Note that human-written method names are not always informative, while a well-trained MNG model has the potential to generate better method names in many cases [17]. Therefore, in addition to parameter sharing among the three tasks in a classic MTL manner, we incorporate a **two-pass deliberation process** shown in Figure 2 into our MTL architecture, to refine method name representations fed into the second-pass code summarization. The mechanism has two main characteristics:

- The decoding processes of code summary decoder and method name decoder are sequential instead of parallel. Only when MNG complete the whole decoding process in the first pass, COS will start its own decoding process in the second pass.
- We make the method name encoder and decoder share the same GRU [22] instance, so that their intermediate states are in the same vector space. Then information from a method name and a generated method name can be fused and refined by weighted addition (or a gate) of context vectors generated by states of method name encoder and decoder during decoding, respectively. The informativeness scores generated by MNIP are normalized and then used as the weights of the fusion gate.

B. Method Name Generation

The MNG task plays two main roles. One is to improve generalization capabilities as a classic auxiliary task in multi-task learning, and the other is to contribute to providing more informative representations of method names. It contains two parts: the method body encoder and the method name decoder.

1) Method Body Encoder: The Method Body Encoder aims to convert the input parsed from method body to the corresponding vector representation. In a neural-network model of text summarization, input of the encoder is the token sequence of source text. Similarly, in code summarization task, the code pieces as input can be treated as word sequence [9]. Researchers extract the word tokens in source code and put them into the encoder as input sequence. Although the text encoder achieved good results in code summarization task via observing meaningful words in code, the structural information of the source code, which is the determining factor of source code's behavior, can not be fully captured.

To address this issue, researchers have proposed various encoders specifically for the structured source code [3], [7], [15]. For example, He et al. [3] proposed a new structure-based traversal (SBT) which flattens the AST and ensures that the words in the code are associated with their AST node type at the same time, so that the SBT encoder can capture both the word semantic information and the code structure semantic information. Although SBT archived excellent results on code summarization task, SBT may have limitations on informaton loss cased by OOV(out Of vocabulary) words. As we all know, complex tokens under camel case rule are widely used in source code and these tokens would produce a large number of OOV words [23]. However, in SBT, we can not make use of such tokens easily.

To tackle this problem, we propose a new advanced structure-based traversal (aSBT) method to traverse ASTs based SBT. Using aSBT, the abstract syntax tree (AST) can be converted to two flattened sequence: the aSBT token sequence and aSBT type sequence. TABLE II shows a simple example of aSBT and TABLE III shows all aSBT types and their corresponding descriptions. As shown in TABLE II, SBT token sequence and SBT type sequence represents the content information and structure information in original SBT sequence respectively.

Once the aSBT token sequence and aSBT type sequence are all ready, we use a embedding layer to convert them to aSBT token embedding sequence. So we can get a method body embeddings $x^b = \{x_1^b, x_2^b, ..., x_m^b\}$ with m tokens, where the upperscript b represents method **b**ody.

TABLE II
EXAMPLE OF ASBT

Source code	storage_client = Client()								
original SBT sequence	(Assign SimpleName_storage_client (Call SimpleName_Client) Call) Assign								
aSBT token sequence	Assign	SimpleName	storage	client	Call	SimpleName	Client	Call	Assign
aSBT type sequence	0	2	3	5	0	2	6	1	1

TABLE III
ASBT TYPE DETAILS

aSBT type	description
0	The beginning of AST node type.
1	The end of AST node type.
2	A single AST node type.
3	The beginning of token.
4	The middle of token.
5	The end of token.

Then a RNN layer is used to encode the embedding sequence into a semantic embedding sequence which encode the significant information needed in decoder. We choose the GRU as our RNN layer, the hidden state of the GRU at each time step t is computed as:

$$z_t = \sigma(W_z[h_{t-1}, x_t]) \tag{1}$$

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \tag{2}$$

$$\hat{h}_t = tanh(W[r_t \cdot h_{t-1}, x_t]) \tag{3}$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hat{h}_t \tag{4}$$

where σ is the sigmoid function $\sigma(x) \in [0, 1]$, h_{t-1} is the previous hidden state, x_t is the current input embedding, and variables W_z, W_r, W are the parameters of GRU. To maximize the use of encoder, we use the last hidden state of the previous encoder as initial state of current encoder. We feed the aSBT embeddings into method body encoder and then get the corresponding method body semantic embedding sequence $h^b = \{h_1^b, h_2^b, ..., h_m^b\}$.

2) Method Name Decoder: The method name decoder aims to convert method body semantic embeddings into the code summary word sequence. We factorize the conditional in equation 5 into a product of word-level method name predictions:

$$p(y_1^n, y_2^n, ..., y_m^n | h^b) = \prod_{j=1}^{m} p(y_j^n | y_{<j}^n, h^b) \tag{5}$$

where probability of each y_t^n[1] is predicted based on all the words that are generated previously (i.e. $y_{<t}^n$) and method body representation h^b. More specifically, the probability is calculated as follows:

[1]The upperscript n represents method **n**ame.

$$\begin{aligned} p(y_t^n | y_{<t}^n, h^b) &= softmax(W_s \cdot tanh(W_t \cdot [s_t^n; c_t^b])) \\ s_t^n &= GRU(y_{t-1}^n, s_{t-1}^n) \\ c_t^b &= attention(s_t^n, h^b) \end{aligned} \tag{6}$$

where s_t^n denotes hidden state of decoder at time step t and c_t^b denotes the contextual information in generating word y_t^n according to different encoder hidden states, which is computed as follows:

$$\begin{aligned} \alpha_{ti}^b &= exp(s_t^n W_a h_i^b) / \sum_j^m exp(s_t^n W_a h_j^b) \\ c_t^b &= \sum_i^m \alpha_{ti}^b h_i^b \end{aligned} \tag{7}$$

C. Code Summarization Model

COS model aims to build the mappings bettween source code and the corresponding summaries. And to generate a more informative summary, COS model takes generated method name, human-written method name and method body as inputs and then introduces a deliberation process to fuze the information among them.

1) Method Name Encoder: Since generated method name and human-written method name are both the abstractions of source code, we believe they can benefit from sharing parameters to promote the capacity of capturing the gist of method name. To this end, we use a shared method name encoder to generate hidden state sequences for both generated method name and human-written method name. Similar to method body encoder, the embedding layer and GRU layer are used to project method name to its corresponding vectors, the vectors encode the semantic information on the method name. Hereafter, the human-written method name vectors $h^n = \{h_1^n, h_2^n, ..., h_m^n\}$ and generated method name vectors $h^n = \{h_1^{gn}, h_2^{gn}, ..., h_m^{gn}\}$ are generated, and the upperscript gn represents **g**enerated method **n**ame.

2) Multiple Attention: Multiple attention aims to capture the important clues from the representations generated by method name encoder and method body encoder. We use the standard attention mechanism as our attention method, which can be described as:

$$w_i = \frac{\exp(f(k_i, q))}{\sum_{i=1}^m \exp(f(k_i, q))} \tag{8}$$

$$attention(q, k, v) = \sum_{i=1}^{m} w_i v_i \tag{9}$$

The inputs of attention mechanism consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, and apply a softmax function to obtain the weights on the values. Here f calculates the similarity between hidden states and we use bi-linear function $f(x, y) = xW_{bi}y$; we use the output s_t^s of GRU layer in summary decoder at step t as the query vector, which will be described with further details in section II-C5. Then we obtain the method body context vector $c^b = attention(s_t^s, h^b, h^b)$, the generated method name context vector $c^{gn} = attention(s_t^s, h^{gn}, h^{gn})$ and the human-written method name context vector $c^n = attention(s_t^s, h^n, h^n)$.

3) Method Name Scorer: Method name scorer predict a scalar score by equation 10, indicating how informative a method name is for the corresponding code summary.

$$score(h) = W_p[h_m^b; h_m] \tag{10}$$

Therefore, we compute the informative scores of human-written method name and generated method name as follows:

$$w^n = score(h^n) \tag{11}$$

$$w^{gn} = score(h^{gn}) \tag{12}$$

4) Method Name Fusion: Method name fusion combines the generated method name context vector and human-written method name context vector with the informativeness score calculated by method name scorer:

$$c^{fn} = \hat{w}^n c^n + \hat{w}^{gn} c^{gn} \tag{13}$$

where $\hat{w}^n$ and $\hat{w}^{gn}$ are normalized informativeness scores calculated by method name scorer.

5) Summary Decoder: Similar to the method name decoder, the code summary decoder learns a conditional probability as:

$$p_{cos}(y_t^s | y_{<t}^s) = softmax(W_s \cdot tanh(W_t \cdot [s_t^s; c_t^b; c_t^{fn}])) \tag{14}$$

with $s_t^s = biGRU(y_{t-1}^s, s_{t-1}^n)$ is the decoder variables for each time step in RNN layer, the upperscript s represents summary.

As seen, the deliberation process mainly involves the operations of feeding generated method name into the second-pass and fusing c^n and c^{gn} into a refined representation c^{fn} according to the predictions of MNIP at each decoding step.

6) Multi-Copy: Finally, we employ a multi-copy component to copy words from source code. We propose the multi-copy component that copies a word w from all the method body, human-written method name and generated method name.

$$\begin{aligned} p_{copy^b} &= \sum_{i:w_i=w} w_{t,i}^b \\ p_{copy^n} &= \sum_{i:w_i=w} w_{t,i}^n \\ p_{copy^{gn}} &= \sum_{i:w_i=w} w_{t,i}^{gn} \end{aligned} \tag{15}$$

The final distribution is a weighted sum of the generation distribution and multi-copy distribution:

$$\gamma_t = sigmoid(W_f[s_t^s; c_t^b; c_t^{fn}] + b_f) \tag{16}$$

$$p = \gamma_t p_{cos} + \frac{1}{3}(1 - \gamma_t)(p_{copy^b} + p_{copy^n} + p_{copy^{gn}}) \tag{17}$$

D. Pre-Training

Since the performance of COS task is highly dependent on the performance of MNG and MNIP tasks, we employ a pre-training process to pre-optimize the parameters of MNG and MNIP models.

In task MNG, we first extract the method name from the source code, so we can get a new training corpus of pairs between source code and method name: $S^{mng} = \{x^{(i)}, y^{(g,i)}\}$. With the given training corpus S^{mng}, the COS task's training objective is to minimize the negative log-likelihood of the training data with the respect to all parameters, as denoted by θ,

$$loss_{mng} = -\sum_{i=1}^{|S^{mng}|} \sum_{j=1}^{|y^{(g,i)}|} logP(y^{(g,i)} | x^{(i)}; y_{<j}^{(g,i)}; \theta) \tag{18}$$

In task MNIP, we use the proportion of words in method name also appearing in the corresponding code summary as the golden informativeness score, so we can get a new MNIP corpus: $S^{mnip} = \{x^{(i)}, y^{(ip,i)}\}$. We minimize a mean-square error criterion over the training set:

$$loss_{mnip} = \frac{1}{|S^{mnip}|} \sum_{i=1}^{|S^{mnip}|} (y^{(ip,i)} - w^{(n,i)})^2 \tag{19}$$

E. Multi-Task Training

Given a training corpus of code-summary pairs: $S^{cos} = \{x^{(i)}, y^{(s,i)}\}$, the training details introduced as follows. With the given training corpus S, the COS task's training objective is to minimize the negative log-likelihood of the training data with the respect to all parameters, as denoted by θ,

$$loss_{cos} = -\sum_{i=1}^{|S^{cos}|} \sum_{j=1}^{|y^{(s,i)}|} logP(y^{(s,i)} | x^{(i)}; y_{<j}^{(s,i)}; \theta) \tag{20}$$

Our final joint learning objective becomes:

$$loss = loss_{cos} + \alpha loss_{mng} + \beta loss_{mnip} \tag{21}$$

III. Experiments

A. Dataset Details

To ensure the generality of our experiment results, we use three datasets to evaluate our approach.

1) Java Summary Dataset: This is the standard Java dataset provided by Leclair et al. [13], which contains over 2.1 million Java methods and associated comments. Following Leclair et al. [1], we split it into training, validation, and testing sets in proportion of 90 : 5 : 5 after shuffling the pairs.

2) Python Summary Dataset.: This dataset is built by Barone et al. [5]. It is a diverse parallel corpus of Python methods with their documentation strings. The dataset contains 113,108 code-comment pairs, and the pairs which can not be parsed by Python3 are removed. We use the same split of training, validation, and testing sets as the original settings in Barone et al. [5], consisting of 109,108 training examples, 2,000 validation examples and 2,000 test examples.

3) Java Method Name Dataset.: This dataset is collected from popular Java projects that have at least 100 stars on GitHub by ourselves, which is only used for pre-training of the MNG task. We build this dataset to see whether pre-training of MNG task on large-scale data can improve the performance. This dataset has more than 30.4 million Java methods and corresponding method names.

B. Experiment Settings

Following Leclair et al. [1], we set the dimensionality of the GRU hidden states, method body embeddings and summary embeddings to 256, 100 and 100, respectively . The method name share the same vocabulary and embedding space with the summary. The hyper-parameter α and β are set to 0.1 and 0,1 since they achieved best performance. The maximum lengths for method name sequences, method body sequences and summary sequences are 10, 300, 13 for Java, and 10, 100 and 20 for Python, each covering at least 90% of the training set. Sequences that exceed the maximum will be truncated and the shorter sequences are padded with zeros. The vocabulary size of the method body, and summary are 50000, 44707 for Java and 50400, 31350 for Python according to Leclair et al. [1] and Yao et al. [4] respectively. For each approach, we computed performance metrics for the model after each epoch against the validation set. Then we chose the model after the epoch with the highest validation performance, and computed performance metrics for this model against the testing set. Adam is used for parameter optimization. The learning rate is set to 0.001. Our model is implemented in PyTorch and trained on Tesla T4.

C. Metrics

Following previous works, we evaluate the performance of DMACOS and baselines based on BLEU4 [24], METEOR [25] and ROUGE-L [26], all of which are widely used in evaluating performance of text generation tasks. BLEU4 score is a popular accuracy-based measure for machine translation. It calculates the similarity between the generated sequence and reference sequence by counting the n-grams that appear in both the candidate sequences and the reference sequence. METEOR measure is the harmonic average of precision and recall, with argument that recall-based measures can be more correlative to manual judgement than accuracy-based measures like BLEU. ROUGE-L takes into account sentence level structural similarity naturally and identifies the longest co-occurring in sequence n-grams automatically.

D. Research Questions

Our motivating intuition is that leveraging method name for two-pass deliberation multi-task learning can improve the results of existing code summarization techniques. Therefore, in our experiment, the major goal is to validate this intuition and find out how much enhancement our technique can bring to various existing techniques. So we try to answer the following research questions to achieve this goal.

1) RQ1: How effective is DMACOS compared to state-of-the-art neural-network-based models?: There are many neural-network-based models for code summarization, and we selected representative ones as baselines and will be described in Section III-E. We evaluated the performance of DMACOS in Java summary dataset and Python summary dataset using metrics described earlier.

2) RQ2: Can DMACOS improves performance of existed encoder-enhaning models?: Researchers in the code summarization field have proposed a variety of neural models, in which many of them tried to design better code encoder to capture more accurate semantics of code. These encoder-enhaning models can be easily equipped with our MTL architecture. To demonstrate the effectiveness of DMACOS in terms of improving encoder-enhaning models, we implemented their DMACOS-equipped versions, and compared the evaluation metric scores with those of their original versions based on the aforementioned metrics.

3) RQ3: How does MTL, deliberation and MNIP affect the performance of DMACOS?: To demonstrate how does the MTL, deliberation and MNIP affect the performance of DMACOS, we removed multi-task learning, two-pass deliberation process and MNIP task from the DMACOS and implement three variant models. We compared these three models based on the aforementioned metrics.

4) RQ4: How does DMACOS perform in the scenarios of poorly-named methods?: Since poorly-defined method names are common in practical software projects, we designed **RQ4** to study how a model performs if the method names are poorly-defined. To simulate the scenario of handling non-informative method names, we replaced method names with a universal special "UNK" token on source code and compared the performance of DMACOS and Dual model on such replaced dataset.

5) RQ5: Can DMACOS improves performance of code summarization further if the MNP pre-training is performed on a larger-scale training dataset?: One major advantage of DMACOS is that it can take advantage of the naturally available large-scale dataset for the MNP task. When answering the first two research questions, we show that DMACOS can already enhance performance of code summarization by pre-training MNP task on the same dataset. **RQ5** tries to investigate whether a larger pre-training dataset for the MNP task can further enhance experiment results. To answer **RQ5**, we first pre-trained a Seq2Seq model which consist of SBT encoder, text encoder and method name decoder on the Java method name dataset for MNP task. Then we initialize the parameters in SBT encoder and text encoder with the pretrained

parameters in baseline model and our model. Finally we fine-tuning these models on the standard Java summary dataset in different dataset size. After that, We compare these models based on the aforementioned metrics.

6) RQ6: How does MNG pre-training perform in the scenarios of cross-programming-language training?: For less popular programming languages, it is often difficult to construct a large training dataset for either MNP or code summarization tasks. Therefore, we designed **RQ6** to study how a model performs if it is pre-trained (for the MNP task) on one programming language (which is more popular), and tuned (for the code summarization task) on another language. To answer this question, we trained a DMACOS-based model on the python dataset with MNP pre-training on the Java dataset.

E. Baselines

We compare our approach with the following and state-of-the-art models as baselines of our evaluation.

- **CodeNN** CodeNN [9] is an end-to-end code summarization approach. They use LSTM to generate summaries given code snippets. At each time step, CodeNN generates a word by applying the attention mechanism, which computes a weighted sum of the code token embeddings.
- **SBT** SBT [3] is a model based on the popular attention-based seq2seq NMT systems. The model takes the structure-based traversal (SBT) representation as input, which converts the abstract syntax tree (AST) to a flattened sequence.
- **Ast-attendgru** Ast-attendgru [1] is an approach which uses both original code text and SBT representation as input. The approach builds a GRU-based-encoder-decoder model with two encoders and one decoder. At each time step of decoding, attention mechanism is applied between the outputs of all the encoders and the decoder.
- **DRL** DRL [4] incorporate an abstract syntax tree structure as well as sequential content of code snippets into a deep reinforcement learning framework. And the BLEU metric score is used as the advantage reward to provide global guidance for explorations.
- **Dual Model** Dual model [12] improve code summarization by introducing a dual code generation task and designing a regularization terms based on the dualities of code summarization and code generation on the probability and attention weights.

IV. Experiment Results

A. Overall Results

Our first research question is: *How effective is MACOS compared to state-of-the-art neural-network-based models?* To answer **RQ1**, We evaluated and compared MACOS against the baselines CodeNN, SBT, Ast-attendgru, DRL and Dual model, among which the Dual model is state-of-the-art code summarization method. TABLE IV reports the performance of the baselines and our model, and our model outperforms all baselines in the three evaluation metrics.

As shown in TABLE IV, the BLEU4, METEOR and ROUGE_L scores of DMACOS are higher than all baselines in both Java and Python datasets. Compared with a reinforcement learning model (DRL), the relative enhancements for BLEU4, METEOR and ROUGE_L on Java and Python dataset are 6.6%, 8.0%, 3.9%, 13.8%, 16.6% and 8.1%, respectively. Compared with a dual learning model (Dual Model), the relative enhancements for BLEU4, METEOR and ROUGE_L on Java and Python dataset are 4.8%, 4.7%, 3.2%, 11.0%, 8.6% and 4.6%, respectively. The results verify that code summarization can be improved significantly by better exploiting the relationship between method code summaries and method names.

B. Applying DMACOS to other models

Our second research question is: *Can DMACOS improves performance of existed encoder-enhaning models?* To answer **RQ2**, we evaluated and compared SBT and Ast-attendgru with their MACOS-equipped versions on the aforementioned metrics. In particular, we plug their encoder into DMACOS by simply replacing the original method body encoder, so that we obtain DMACOS(SBT) and DMACOS(Ast-attendgru).

The seventh and eighth rows of results in TABLE IV show that DMACOS led to significant enhancement on SBT and Ast-attendgru, e.g., with the relative increment on BLEU4 in Java and Python datasets as 27.4%, 31.6%, 9.7% and 44.6% respectively. It is notable that applying DMACOS method to Ast-attendgru even achieves a better performance than Dual Model, the latest competitive approach.

C. Ablation Study

Our third research question is: *How does MTL, deliberation and MNIP affect the performance of DMACOS?* To answer **RQ3**, We obtain three model variants with a subset of DMACOS's key components: (1) **DMACOS w/o MTL** with the entire MTL design removed, (2) **DMACOS w/o two-pass** with only MNG as an auxiliary task in a classic single-pass MTL architecture, and (3) **DMACOS w/o MNIP** that employ the two-pass mechanism that feeding generated method name information of the first pass directly into the summary decoder without the refining process guided by MNIP. The performance of the three DMACOS variants is also reported in TABLE IV, from which we have the following observations:

1) It is as expected that removing all MTL components makes the performance degrade significantly (e.g., 11.1 of DMACOS w/o MTL v.s. 12.9 of DMACOS in terms of BLEU score on Java dataset).
2) Compared with DMACOS w/o two-pass, DMACOS w/o MNIP gain a relative improvement for BLEU4, METEOR, and ROUGE_L of 2.5%, 3.1%, 5.2% (Java dataset) and 5.0%, 4.4%, and 1.1% (Python dataset), verifying the effectiveness of our two-pass design even with only task MNG.
3) Compared with DMACOS w/o MNIP, full DMACOS achieves further improvement (e.g., a relative BLEU4 improvement of 3.2% and 3.9% on Java and Python

TABLE IV
EXPERIMENTAL RESULTS ON THE JAVA AND PYTHON DATASETS. RELATIVE IMPROVEMENTS WE CONCERNED ARE ALL APPLIED T-TEST AND ALL P-VALUES ARE SMALLER THAN 0.01, INDICATING SIGNIFICANT INCREASES.

Approaches	Java			Python		
	BLEU4	METEOR	ROUGE-L	BLEU4	METEOR	ROUGE-L
CodeNN	6.85	13.9	32.1	6.84	13.1	28.3
SBT	7.70	15.2	35.0	7.90	13.4	29.9
Ast-attendgru	11.3	17.9	39.5	8.71	16.5	34.8
DRL	12.1	18.6	40.7	11.6	16.3	35.6
Dual Model	12.3	19.2	41.2	11.9	17.5	36.8
DMACOS	**12.9**	**20.1**	**42.3**	**13.2**	**19.4**	**38.5**
DMACOS(SBT)	9.81	18.0	39.4	10.4	17.1	35.7
DMACOS(Ast-attendgru)	12.4	20.2	44.4	12.6	17.6	36.5
DMACOS w/o MTL	11.1	18.6	38.9	8.70	16.1	33.2
DMACOS w/o two-pass	12.2	19.1	38.8	12.1	18.1	37.9
DMACOS w/o MNIP	12.5	19.7	40.8	12.7	18.9	38.3
pre-trained DMACOS	**13.7**	**21.0**	**45.9**	**14.1**	**19.7**	**39.5**

TABLE V
EXPERIMENTAL RESULTS ON THE JAVA AND PYTHON DATASET WITH METHOD NAME MASKED.

Approaches	Java			Python		
	BLEU4	METEOR	ROUGE-L	BLEU4	METEOR	ROUGE-L
Dual	6.34	15.1	32.7	4.78	12.1	25.1
DMACOS w/o MTL	5.75	12.4	29.6	4.87	11.6	25.2
DMACOS w/o two-pass	4.16	10.1	24.7	5.92	10.4	24.7
DMACOS w/o MNIP	7.39	16.1	33.3	7.11	11.9	27.5
DMACOS	**8.21**	**17.6**	**37.0**	**8.25**	**13.3**	**29.6**

dataset respectively) via the MNIP-guided refinement on method name representations. This ablation study clearly demonstrates the effectiveness of all designs involving introducing and utilizing MNG and MNIP.

D. Impact of method name informativeness

Our fourth research question is: *How does DMACOS perform in the scenarios of poorly-named methods?* To answer **RQ4**, we replaced method names with a universal special "UNK" token in the testing dataset to simulate the scenario of handling non-informative method names. Under this experiment setting, the performance of DMACOS, its variants, and Dual Model are shown in Table V, from which we have three observations.

1) Significant performance degradation occurs in all models, indicating the significance of method names in code summarization.
2) Interestingly, the model using a classic MTL architecture without the two-pass mechanism (DMACOS w/o two-pass) performs worse than the model without MTL architecture (DMACOS w/o MTL), e.g., on the Java dataset. This observation reflects the side effects of being too dependent on well-defined method names.
3) The degradation of DMACOS is much slighter than that of DMACOS w/o two-pass (e.g., about one-third v.s. two-third of relative decrement in terms of BLEU score on the Java dataset), showing that DMACOS can alleviate the impact of non-informative method names better.
4) Comparison results among models (e.g., DMACOS v.s. DMACOS w/o MNIP, DMACOS v.s. Dual Model) generally follow a similar pattern with the main results on standard experiments.

Note that to evaluate the impact of method name informativeness on DMACOS, another straightforward way is dividing samples in the test set into some subsets based on their informativeness. However, we find the quality of code summaries in each subset diversify enormously, making it difficult to make fair comparisons among results on these subsets, though we find that DMACOS always outperforms the Dual model on each subset.

As a further qualitative analysis, recapping the case illustrated in Figure 1 in Section I, here we feed those non-informative method names into DMACOS. Summaries generated by our DMACOS and Dual Model are shown in Figure 3, and DMACOS's output is obviously more consistent with human-written summaries.

E. Pre-Training on a Larger Dataset

Our fifth research question is: *Can DMACOS improves performance of code summarization further if the MNP pre-training is performed on a larger-scale training dataset?* To answer **RQ5**, we first performed pre-training for the MNP task on the large dataset we collected from Github. Then we performed the multi-task learning on the standard dataset of Java language. This technique is also called fine-tuning in deep learning community.

The results of the model with MNG pre-trained (pre-trained DMACOS) are shown in the part of Java dataset on last row in TABLE IV. The results show that on all datasets and all metrics, pre-trained DMACOS performs better than DMACOS (e.g., 13.7 v.s. 12.9 in terms of BLEU score on the Java

Human-written Summary: return the given decimal number formatted for a specific locale.

Method Name	Generated By Dual	Generated By DMACOS
format_decimal	return a format number according to the format pattern	return the decimal number formatted for locale.
f_dec	creates an output quoted pattern according to locale a pattern as a quoted string pattern.	return the given decimal number formatted.
formatdecimal	returns a given format pattern according to the pattern.	return the given character formatted string pattern.
formt_decimal	return a local number format according to the locale rules in the format.	return the given decimal number formatted.

Fig. 3. Summaries generated by Dual Model and DMACOS with different method names. The corresponding source code is shown in Figure 1.

summary dataset). The results also show that MNP large-scale corpus has a large tolerance to noise, allowing us to utilize the large-scale corpus without carefully designing filtering rules, which is an advantage brought by self-labeled dataset.

F. Cross-Programming-Language Training

Our sixth research question is: *How does MNG pre-training perform in the scenarios of cross-programming-language training?* To answer **RQ6**, we trained a MACOS-based model on the python dataset with pre-trained MNP from corpus of Java. The results of the model with MNG pre-trained (pre-trained DMACOS) are shown in the part of Python dataset on last row in TABLE IV. The results indicate that knowledge captured by the MNG task can be utilized even across programming languages. The reason is that the text and structure semantics within the code do have similarities among different programming languages and the MNG task can characterize them well via large-scale corpus to improves performance.

G. Threats to Validity

One threat to the construction validity of our experiment is that we use automatic metrics to evaluate the quality of summaries. All of these metrics may have limitations on simulating the quality of generated code summary. However, since human-based evaluation is not scalable and can still be subjective, these metrics are widely used in prior research efforts on text generation (including code-summary generation). To reduce this threat, we used multiple metrics in our evaluation to avoid limitations of a specific metric affecting our results. To further reduce this threat, we plan to perform human-based evaluation in the future and combine human-based evaluation metrics with automatic evaluation metrics.

The major threat to internal validity is that we use the first sentence of code documentation as our ground truth. This is also a common practice in prior research efforts on code summarization [1]. This approach will inevitably introduces noise into the data, e.g., mismatches between methods and the one-sentence summarization. It is necessary to investigate how to build a better parallel corpus. And we believe that with the in-depth application of neural network in code summarization, we will be able to build models with excellent performance on much fewer parallel corpora, and MACOS is a solid step toward this direction. Another threat to internal validity is that we did not perform cross-validation. We attempt to mitigate this risk by using random samples to split the training/validation/testing sets, a different split could result in different performance. This risk is common among summarization experiments due to very high training computation costs.

The major threat to external validity is that our experiment results may be applicable only to the dataset we used in our experiment. To reduce this threat, we used two very large evaluation dataset in two different popular programming langauges: Java and Python. To further reduce this threat, in the future, we plan to perform more experiments on more datasets in other programming languages.

V. Related Work

A. Code Summarization

Automatic code summarization now is an important and rapidly-growing research topic in the community of software engineering and natural language processing. For traditional techniques, we direct refers to a comprehensive survey by Nazar et al. [27]. Since the first neural model of code summarization was proposed by Iyer et al. [9], we have witnessed the introduction of the latest neural technologies for text generation tasks (e.g., machine translation and text summarization) into this research field recently. For example, Wei et al. [12] and Ye et al. [11] introduced the code generation task to improve code summarization task via dual learning [28]. Hu et al. [6] proposed an API sequence encoder to assist code summarization by transferring learning of API knowledge. Yao et al. [4] adopted Tree-RNN and reinforcement learning to enhance BLEU score of code summaries. Chen et al. [10] designed a neural framework for code summarization and code retrieval based on Variational Auto Encoders (VAEs).

B. Method Name Prediction

Method name prediction problem can be seen as structured prediction problem. Given method codes, a sequence of method name is needed to be predicted. Regarding both method code and method name as a sequence, the seq2seq paradigm [29] can be used to model the problem. Allamanis

et al. [30] use a logbilinear context model to generate method names. They define a local context capturing tokens around the current token, and a global context which is composed of features from the method code. As mentioned above, Allamanis et al. [14] propose an attentional neural network that employs convolution on the input tokens to detect features for method name prediction.The abstract syntax tree (AST) of method code contains meaningful structural properties, thus models based on tree structures like Tree-LSTM [31] can encode the method code in the form of AST. To better encode source code, Alon et al. [7] present code2seq model, which leverages the compositional paths in AST and uses attention to select the relevant paths while decoding. In a subsequent paper [15], the path representation is computed using Long Short Term Memory network (LSTM) instead of a single layer neural network.

C. Code Representation

Code representation and code summarization are two closely related studies. The code representation studies how to generate word vectors for each token or an overall representation for a piece of code. Harer et al. [32] use word2vec to generate word embedding for C/C++ tokens for software vulnerability prediction.The token embedding is used to initialize a TextCNN model for classification. Xie et al. [33] and Zhang et al. [34] build a code knowledge graph and learn code context representation based on this knowledge graph. The learned representations are used to recover the missing links between issues and commits. Mou et al. [35] learn distributed vector representations using custom convolutional neural networks to represent features of snippets of code, then they assume that student solutions to various coursework problems have been intermixed and seek to recover the solution-to-problem mapping via classification. Li et al. [36] 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. Alon et al. [15] compute Java method embeddings by decomposing code to a collection of paths between two leaf nodes in its abstract syntax tree, and learning the atomic representation of each path simultaneously with learning how to aggregate a set of them. The method embedding is used to predicting method names.

D. Multi-Task Learning

MTL is heavily used in machine learning and natural language processing tasks. MTL learning aims to help improving the learning of a model by leveraging the domain-specific knowledge contained in the training signals of related tasks [37]. Usually, relatedness among tasks are learned in two ways in deep neural networks: hard parameter sharing and soft parameter sharing of hidden layer [38]. Hard parameter sharing MTL was first proposed in [39], which shares the hidden layer between all tasks and keep task-specific output layers. Collobert et al. [40] describe a single convolutional neural network architecture trained jointly on NLP tasks such as part-of-speech tags, chunks, named entity tags, and semantic roles. Zheng et al. [41] propose a module in which all tasks share the same sentence representation and each task can select the task-specific information from the shared sentence representation with attention mechanism. On the other hand, each task in soft parameter MTL contains its own model and parameters, and the parameters are encouraged to be similar with regularization. Misra et al. [42] connect two separate networks in a soft parameters sharing way. Then the model leverages a unit called cross-stitch to determine how to combine the knowledge learned in other related tasks to task-specific networks.

We first introduced MTL into the study of code summarization. Hu et al. [6] use a fine-tuning approach; however, their pre-training is mainly used to capture features (e.g., API call sequences) from the input data of the same task, rather than share knowledge of different tasks. They declared their work as a transfer learning approach, but we believe it is more about designing better code features, since datasets used for pre-training and fine-tuning are generally the same. We believe that our research provides a reference for code learning tasks based on deep learning. The method of mining the association between different tasks and large-scale corpus pre-training has great potential in the field of program representation and comprehension.

VI. Conclusions and future work

In this paper, we leverage method name generation (MNG) and method name informativeness prediction (MNIP) to improve code summarization via a two-pass deliberation multi-task learning approach. In our approach DMACOS, MNG not only serves as a highly-related auxiliary task that provides beneficial inductive bias, but also helps yielding better representations of method names with the useful guidance from task MNIP in a two-pass way. Empirical results show DMACOS could improve model performance and alleviate the impact of non-informative method names.

The novelty of DMACOS mainly lies in the idea of introduction and effective utilization of task MNG and MNIP. It is easy to be applied to other code summarization models as demonstrated in the experiments as well as other potential scenarios of text generation. Taking summarization on internet news as an example, we can introduce generation and informativeness prediction of news headlines as auxiliary tasks in the same manner as DMACOS, of which the headline generation is also a self-supervised task. More sophisticated deliberation designs can so be incorporated into our architecture for these scenarios, which is our future work.

References

[1] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[2] P. Mcburney and C. Mcmillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

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

[4] Y. Wan, Z. Zhao, M. Yang, G. Xu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *the 33rd ACM/IEEE International Conference*, 2018.

[5] 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.

[6] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 2018, pp. 2269–2275.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR*, 2019.

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

[9] 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.

[10] 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, ASE*, 2018, pp. 826–831.

[11] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," *CoRR*, vol. abs/2002.10198, 2020.

[12] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 6559–6569.

[13] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Jun. 2019, pp. 3931–3937.

[14] 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.

[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[16] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 294–317.

[17] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1–12.

[18] Y. Xia, F. Tian, L. Wu, J. Lin, T. Qin, N. Yu, and T. Liu, "Deliberation networks: Sequence generation beyond one-pass decoding," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 1784–1794.

[19] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, 2019, pp. 4171–4186.

[20] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019*, 2019, pp. 5754–5764.

[21] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019.

[22] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2014, pp. 1724–1734.

[23] S. Zhang, R. Xie, W. Ye, and L. Chen, "Keyword-based source code summarization," *Journal of Computer Research and Development*, vol. 57, no. 9, p. 1987, 2020.

[24] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA.*, 2002, pp. 311–318.

[25] S. Banerjee and A. Lavie, "METEOR: an automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005*, 2005, pp. 65–72.

[26] C.-Y. Lin, "Rouge: A packagefor automatic evaluation of summaries," in *ProceedingsofWorkshop on Text Summarization Branches Out, Post2Conference Workshop of ACL*, 2004.

[27] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 883–909, 2016.

[28] D. He, Y. Xia, T. Qin, L. Wang, N. Yu, T. Liu, and W. Ma, "Dual learning for machine translation," in *Advances in Neural Information Processing Systems 29*, 2016, pp. 820–828.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, 09 2014.

[30] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[31] S. T. Kai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *Computer Science*, vol. 5, no. 1, p. : 36., 2015.

[32] J. A. Harer, L. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," 02 2018.

[33] R. Xie, L. Chen, W. Ye, Z. Li, T. Hu, D. Du, and S. Zhang, "Deeplink: A code knowledge graph based deep learning approach for issue-commit link recovery," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, 2019, pp. 434–444.

[34] J. Zhang, R. Xie, W. Ye, Y. Zhang, and S. Zhang, "Exploiting code knowledge graph for bug localization via bi-directional attention," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 219–229.

[35] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[36] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[37] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[38] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[39] R. A. Caruana, "Multitask learning: A knowledge-based source of inductive bias," *Machine Learning Proceedings*, vol. 10, no. 1, pp. 41–48, 1993.

[40] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[41] R. Zheng, J. Chen, and X. Qiu, "Same representation, different attentions: Shareable sentence representation learning from multiple tasks," *arXiv preprint arXiv:1804.08139*, 2018.

[42] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert, "Cross-stitch networks for multi-task learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3994–4003.