TranS$^3$: A Transformer-based Framework for Unifying Code Summarization and Code Search

Wenhua Wang  
11760006@mail.sustech.edu.cn  
Southern University of Science and Technology  
P.O. Box 1212  
Shenzhen, Guangzhou, China 43017-6221

Zhengran Zeng  
11612527@mail.sustech.edu.cn  
Southern University of Science and Technology  
P.O. Box 1212  
Shenzhen, Guangzhou, China 43017-6221

Yuqun Zhang  
zhangyq@sustech.edu.cn  
Southern University of Science and Technology  
P.O. Box 1212  
Shenzhen, Guangzhou, China 43017-6221

Guandong Xu  
Guandong.Xu@uts.edu.au  
University of Technology, Sydney  
Sydney, NSW, Australia 43017-6221

ABSTRACT

Code summarization and code search have been widely adopted in software development and maintenance. However, few studies have explored the efficacy of unifying them. In this paper, we propose TranS$^3$, a transformer-based framework to integrate code summarization with code search. Specifically, for code summarization, TranS$^3$ enables an actor-critic network, where in the actor network, we encode the collected code snippets via transformer- and tree-transformer-based encoder and decode the given code snippet to generate its comment. Meanwhile, we iteratively tune the actor network via the feedback from the critic network for enhancing the quality of the generated comments. Furthermore, we import the generated comments to code search for enhancing its accuracy. To evaluate the effectiveness of TranS$^3$, we conduct a set of experimental studies and case studies where the experimental results suggest that TranS$^3$ can significantly outperform multiple state-of-the-art approaches in both code summarization and code search and the study results further strengthen the efficacy of TranS$^3$ from the developers’ points of view.

1 INTRODUCTION

Code summarization and code search have become increasingly popular in software development and maintenance [1–6], because they can help developers understand and reuse billions of lines of code from online open-source repositories and thus significantly enhance software development and maintenance process [6]. In particular, since much of the software maintenance effort is spent on understanding the maintenance task and related software source code [7], effective and efficient documentation is quite essential to provide high-level descriptions of program tasks for software maintenance. To this end, code summarization aims to automatically generate natural language comments for documenting code snippets [8]. On the other hand, over years various open-source and industrial software systems have been rapidly developed where the source code of these systems is typically stored in source code repositories. Such source code can be treated as important reusable assets for developers because they can help developers understand how others addressed similar problems for completing their program tasks, e.g., testing [9–15], fault localization [16–18], program repair and synthesis [19–22], in multiple software development domains [23–25]. Correspondingly, there also raises a strong demand for an efficient search process through a large codebase to find relevant code for helping programming tasks. To this end, code search refers to automatically retrieving relevant code snippets from a large code corpus given natural language queries.

The recent research progress towards code summarization and code search can mainly be categorized to two classes: information-retrieval-based approaches and deep-learning-based approaches. To be specific, the information-retrieval-based approaches derive the natural language clues from source code, compute and rank the similarity scores between them and source code/natural language queries for recommending comments/search results [2, 4, 26, 27]. The deep-learning-based approaches use deep neural networks to encode source code/natural language into a hidden space, and utilize neural machine translation models for generating code comments and computing similarity distance to derive search results [5, 6, 28–30].

Based on the respective development of code summarization and code search techniques, we infer that developing a unified technique for optimizing both domains simultaneously is not only mutually beneficial but also feasible. In particular, on one hand, since the natural-language-based code comments can reflect program semantics to strengthen the understanding of the programs [1], adopting them in code search can improve the matching process with natural language queries [6]. Accordingly, injecting code comments for code search is expected to enhance the search results [31]. On the other hand, the returned search results can be utilized as an indicator of the accuracy of the generated code comments to guide their optimization process. Moreover, since code summarization and code search can share the same technical basis as mentioned above, it can be inferred that it is feasible to build a framework to unify and advance both the domains. Therefore, it is essential to integrate code summarization with code search.

Although integrating code summarization with code search can be promising, there remains the following challenges that may compromise its performance: (1) state-of-the-art code summarization techniques render inferior accuracy. According to the recent advances in code summarization [26, 32, 33], the accuracy of the generated code comments appears to be inferior for real-world
applicability (around 20% in BLEU-1 with many well-recognized benchmarks). Integrating such code comments might lead to inaccuracies of matching natural language queries and further compromise the performance of code search. (2) how to effectively and efficiently integrate code summarization with code search remains challenging. Ideally, the goal of integrating code summarization with code search is to optimize the performance of both domains rather than causing trade-offs. Moreover, such integration is expected to introduce minimum overhead. To this end, it is essential to propose an effective and efficient integration approach.

To tackle the aforementioned problems, in this paper, we propose a framework, namely TransS, for optimizing both code summarization and code search based on a recent NLP technique—transformer [34]. Unlike the traditional CNN-based approaches that suffer from long-distance dependency problem [35] and RNN-based approaches that suffer from excessive load imposed by sequential computation [36], transformer advances in applying the self-attention mechanism which can parallelize the computation and preserve the integral textual weights for encoding to achieve the optimal accuracy of text representation [34].

TransS consists of two components: the code summarization component and code search component. Specifically, the code summarization component is initialized by preparing a large-scale corpus of annotated < code; comment > pairs to record all the code snippets with their corresponding comments as the training data. Next, we extract the semantic granularity of the training programs for constructing a tree-transformer to encode the source code into hidden vectors. Furthermore, such annotated pair vectors are injected into our deep reinforcement learning model, i.e., the actor-critic framework, for the training process, where the actor network is a formal encoder-decoder model to generate comments given the input code snippets; and the critic network evaluates the accuracy of the generated comments according to the ground truth (the input comments) and give feedback to the actor network. At last, given the resulting trained actor network and a code snippet, its corresponding comment can be generated. Given a natural language query, the code search component is launched by encoding the natural language query, the generated code comments, and the code snippets into the vectors respectively via transformer and tree-transformer. Next, we compute similarity scores between query/code vectors and query/comment vectors for deriving and optimizing their weighted scores. Eventually, we rank all the code snippets according to their scores for recommending the search results. The underlying transformer in TransS can enhance the quality of the generated code and thus strengthen the code search results by importing the impact from the generated comments. Moreover, since the code search component applies the encoder trained by the code summarization component without incurring extra training process, its computing overhead can be maintained minimum.

To evaluate the effectiveness and efficiency of TransS, we conduct a set of experiments based on the GitHub dataset in [37] which includes over 120,000 code snippets of Python functions and their corresponding comments. The experimental results suggest that TransS can outperform multiple state-of-the-art approaches in both code summarization and code search, e.g., TransS can significantly improve the code summarization accuracy from 47.2% to 141.6% in terms of BLEU-1 and the code search accuracy from 5.1% to 28.8% in terms of MRR compared with the selected state-of-the-art approaches. In addition, we also conduct case studies for both code summarization and code search where the study results further verify the effectiveness of TransS.

In summary, the main contributions of this paper are listed as follows:

- **Idea.** To the best of our knowledge, we build the first transformer-based framework for integrating code summarization and code search, namely TransS, that can optimize the accuracy of both domains.
- **Technique.** To precisely represent the source code, we design a transformer-based encoder and a tree-transformer-based encoder for encoding code and comments by injecting the impact from the semantic granularity of well-formed programs.
- **Evaluation.** To evaluate TransS, we conduct a substantial number of experiments based on real-world benchmarks. The experimental results suggest that TransS can outperform several existing approaches in terms of accuracy of both code summarization and code search. In addition, we also conduct empirical studies with developers. The results suggest that the quality of the generated comments and search results are widely acknowledged by developers.

The reminder of this paper is organized as follows. Section 2 illustrates some preliminary background techniques. Section 3 gives an example to illustrate our motivation for unifying code summarization and code search. Section 4 elaborates the details of our proposed approach. Section 5 demonstrates the experimental and study results and analysis. Section 6 introduces the threats to validity. Section 7 reviews the related work. Section 8 concludes this paper.

## 2 BACKGROUND

In this section, we present the preliminary background techniques relevant to TransS, including language model, transformer, and reinforcement learning, which are initialized by introducing basic notations and terminologies. Let \( x = (x_1, x_2, \ldots, x_{|x|}) \) denote the code sequence of one function, where \( x_i \) represents a token of the code, e.g., "def", "fact", or "if" in a Python statement "def fact(i)". Let \( y = (y_1, y_2, \ldots, y_{|y|}) \) denote the sequence of the generated comments, where \( y_j \) denotes the sequence length. Let \( T \) denote the maximum step of decoding in the encoder-decoder framework. We use notation \( y_{1:m}, y_{1:m} \) to represent the comment subsequence \( y_1, \ldots, y_m \) and \( D = \{(x_N, y_N)\} \) as the training dataset, where \( N \) is the size of training set.

### 2.1 Language Model

A language model refers to the decoder of neural machine translation which is usually constructed as the probability distribution over a particular sequence of words. Assuming such sequence with its length \( T \), the language model defines \( p(y_{1:T}) \) as its occurrence probability which is usually computed based on the conditional probability from a window of \( n \) predecessor words, known as n-gram [38], as shown in Equation 1.
Figure 1: The Transformer Model Architecture.

\[
p(y_{t:T}) = \prod_{t=1}^{T} p(y_t|y_{1:t-1}) \approx \prod_{t=1}^{T} p(y_t|y_{t-(n-1):t-1})
\]  

(1)

While the n-gram model can only predict a word based on a fixed number of predecessor words, a neural language model can use predecessor words with longer distance to predict a word based on deep neural networks which include three layers: 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 individual words from the input sentence, and predicts the subsequent word in turn. For the word \( y_t \), the probability of its subsequent word \( y_{t+1} \), \( p(y_{t+1}|y_{1:t}) \) can be computed as in Equation 2:

\[
p(y_{t+1}|y_{1:t}) = g(h_t)
\]  

(2)

where \( g \) is a stochastic output layer (e.g., a softmax for discrete outputs) that generates output tokens with the hidden state \( h_t \), computed as Equation 3:

\[
h_t = f(h_{t-1}, w(x_t))
\]  

(3)

where \( w(x_t) \) denotes the weight of the token \( x_t \).

2.2 Transformer

Many neural machine translation approaches integrate the attention mechanism with sequence transduction models for enhancing the accuracy. However, the encoding networks are still exposed with challenges. To be specific, the CNN-based encoding networks are subjected to long-distance dependency issues and the RNN-based encoding networks are subjected to the long-time computation. To address such issues, transformer [34] is proposed to effectively and efficiently improve the sequence representation by adopting the self-attention mechanism only. Many transformer-based models e.g., BERT [39], ERNIE [40], XLNET [41], have been proposed and verified to dramatically enhance the performance of various NLP tasks such as natural language inference, text classification, and retrieval question answering [42, 43].

Transformer consists of \( N \) identical layers where each layer consists of two sub-layers. The first sub-layer realizes a multi-head self-attention mechanism, and the second sub-layer is a simple, position-wise fully connected feed-forward neural network, as shown in Figure 1. Note that the output of the first sub-layer is input to the second sub-layer and the outputs of both the sub-layers need to be normalized prior to the subsequent process.

2.2.1 Self-attention Mechanism. The attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

The input consists of queries, keys and values of the dimension \( d_k \). Accordingly, transformer computes the dot products of the query with all keys, divides each resulting element by \( \sqrt{d_k} \), and applies a softmax function to obtain the weights on the values. In practice, we simultaneously compute the attention function on a set of queries which are packed together into a matrix \( Q \). In addition, the keys and values are also packed together into matrices \( K \) and \( V \). Therefore, the matrix of outputs can be computed as:

\[
\text{Attention}(Q, K, V) = \text{softmax}\left( \frac{QK^T}{\sqrt{d_k}} \right)V
\]  

(4)

Instead of implementing a single attention function, transformer adopts a multi-head attention which allows the model to jointly attend to information from different representation subspaces at different positions.

The self-attention mechanism derives the relationships between the current input token and all the other tokens to determine the current token vector for the final input representation. By taking advantage of the overall token weights, such mechanism can dramatically alleviate the long-distance dependency problem caused by the CNN-based transduction models, i.e., compromising the contributions of the long-distance tokens. Moreover, the multi-head self-attention mechanism can parallelize the computation and thus resolve the excessive computing overhead caused by the RNN-based transduction models which sequentially encode the input tokens.

2.2.2 Position-wise Feed-Forward Neural Network. In addition to multi-head self-attention sub-layers, each of the layers contains a fully connected feed-forward neural network, which is applied to each position separately. Since transformer contains no recurrence or convolution, in order to utilize the order of the sequence, transformer injects “positional encodings” to the input embedding.

Since transformer has been verified to be dramatically effective and efficient in encoding word sequences, we infer that by representing code as a sequence, transformer can also be expected to excel in the encoding efficacy. Therefore, in Trans3, we adopt transformer as the encoder.

2.3 Reinforcement Learning for Code Summarization

In code summarization, reinforcement learning (RL) [44] refers to interacting with the ground truth, learning the optimal policy from the reward signals, and generating texts in the testing phase. It 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 [45]. 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. Such text generation process can be viewed as a Markov Decision Process (MDP) \( (state, action, policy, reward) \). Specifically in the MDP settings, \( state \) \( s_t \) at time \( t \) consists of the code snippets \( x \) and the predicted words \( y_t, y_{t+1}, \ldots, y_T \). The action space is defined as the dictionary \( Y \) where the words are drawn, i.e., \( y_t \in Y \). Correspondingly, the state transition function \( P \) is defined as \( s_{t+1} = \{s_t, y_t\} \), where the action \( y_t \) becomes a part of the subsequent state \( s_{t+1} \) and the reward \( r_{t+1} \) can be derived. The objective of the generation process is to find a policy that iteratively maximizes the expected reward of the generated sentence sampled from the model’s policy, as shown in Equation 5,

\[
\max_\theta L(\theta) = \max_\theta \sum_{y_{t-1}, \ldots, y_T} [R(\hat{y}, x)]
\]

where \( \theta \) is the parameter to be learned, \( D \) is the training set, \( \hat{y} \) denotes the predicted actions/words, and \( R \) is the reward function.

To learn the policy, many approaches have been proposed, which are mainly categorized into two classes [46]: (1) the policy-based approaches (e.g., Policy gradients [47]) which optimize the policy directly via policy gradient and (2) the value-based approaches (e.g., Q-learning [48]) which learn the Q-function, and at 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 suffer from a bias issue [49]. To address such issues, the Actor-Critic learning approach is proposed [50] to combine the strengths of both policy- and value-based approaches where the agent chooses an action according to the probability of each action and the critic assigns the value to the chosen action for speeding up the learning process for the original policy-based approaches.

In this paper, we adopt the actor-critic learning model for code summarization of TransS

### 3 ILLUSTRATIVE EXAMPLE

In this section, we use a sample Python code snippet to illustrate our motivation for unifying code summarization and code search. Figure 2 shows the Python code snippet, the comment generated by our approach and its associated natural language query in our dataset. Traditional code search approaches usually compute the similarity scores of the query vector and the code snippet vectors for recommending and returning the relevant code snippets. On the other hand, provided the comment information, it is plausible to enhance the code search results by enabling an additional mapping process between the query and the comments corresponding to the code snippets. For example, in Figure 2, given the query “get the recursive list of target dependencies”, although the code snippet can provide some information such as “dependencies”, “target”, which might be helpful for being recommended, its efficacy can be compromised due to the disturbing information such as “dicts”, “set”, “pending” in the code snippet. It is expected to enhance the search result by integrating the comment information during the searching process when it has the identical “target”, “dependencies” with the query. To this end, we infer that a better code search result can be expected if high-quality comment information can be integrated in the code search process.

### 4 THE APPROACH OF TRANS

We formulate the research problem of integrating code summarization with code search as follows:

- First, we attempt to find a policy that generates a sequence of words \( y = (y_1, y_2, \ldots, y|y|) \) from dictionary \( Y \) to annotate the code snippets in the corpus as their comments. Next, given a natural language query \( x = (x_1, x_2, \ldots, x|k|) \), we aim to find the code snippets that can satisfy the query under the assistance of the generated comments.

To address such research problem, we propose TransS with its framework shown in Figure 3, where Figure 3(a) presents the code summarization part and Figure 3(b) presents the code search part.

#### 4.1 Transformer- and Tree-Transformer-based Encoder

In TransS, we utilize transformer to build the encoder. Specifically, we develop the transformer-based encoder to encode the comments, the query and each program statement. Moreover, we develop a tree-transformer-based encoder that exploits the semantic granularity information of programs for enhancing the program encoding accuracy.

**Transformer-based Encoder.** The transformer-based encoder is initialized by embedding the input tokens into vectors via word embedding [51]. Specifically, we tokenize the natural language comments/questions based on their intervals and the code based on a set of symbols, i.e., \( \{\ldots, \cdot, : *, ()! - \} - \) (space). Next, we apply word embedding to derive each token vector in one input sequence. Furthermore, for each token vector \( x_i \) we derive its representation according to the self-attention mechanism as follows:

1. deriving the query vector \( q_i \), the key vector \( k_i \), and the value vector \( v_i \) by multiplying \( x_i \) with a randomly-generated matrix, (2) computing the scores of \( x_i \) from all the input token vectors by the dot product of \( q_i \cdot k_j \), where \( i \in [1, n] \) and \( n \) denotes the number of input tokens, (3) dividing the scores of \( x_i \) by \( \sqrt{d_k} \) where \( d_k \) denotes the dimension number of \( k_i \) and normalizing the results by softmax
to obtain the weights (contributions) of all the input token vectors, (4) multiplying such weights and their corresponding value vectors to obtain an interim vector space $v'$, and (5) summing all the vectors in $v'$ for deriving the final vector of $x_i, z_i$. As a result, all the token vectors are input to the feed-forward neural network to obtain the final representation vector of the input sequence of natural language comment.

We use Figure 4 as an example to illustrate how the transformer-based encoder works, where the token vectors of "Software" and "Engineering" are embedded as $x_1$ and $x_2$ respectively. For $x_1$, its corresponding $q_1$, $k_1$, and $v_1$ are derived in the beginning. Next, its scores from all the token vectors, i.e., $x_1$ and $x_2$, can be computed by $q_1 \cdot k_1 (112)$ and $q_1 \cdot k_2 (96)$. Assuming $d_k$ is 64, by dividing the resulting dot products by $\sqrt{d_k}$ and normalizing, the weights of $x_1$ and $x_2$ can be computed as 0.88 and 0.12. At last, $z_1$ can be derived by $0.88v_1 + 0.12v_2$.

**Tree-Transformer-based Code Encoder.** It can be observed that well-formed source code can reflect the program semantics through its representations, e.g., the indents of Python. In general, in a well-formed program, the statement with fewer indents tends to indicate more abstracted semantics than the one with longer indents. Therefore, we infer that incorporating the indent-based semantic granularity information for encoding can inject program semantics for program comprehension and thus be promising to enhance the encoding accuracy. Such injection can potentially be advanced when leveraging transformer. In particular, in addition to the original self-attention mechanism which determines the token vector score by only importing the token-level weights, statement-level impacts can be injected by analyzing statement indents, obtaining the semantic hierarchy of the code, and realizing the hierarchical encoding process.

In this paper, we design a tree-transformer-based encoder that incorporates indent-based semantic granularity for encoding programs. Firstly, we construct an ordered tree according to the indent information of a well-formed program. In particular, by reading the program statements in turn, we initialize the tree by building the root node out of the function definition statement. Next, we iteratively label each of the subsequent statements with an indent index assigned by counting the indents such that the statements with the same indent index $i$ are constructed as the ordered sibling nodes and the preceding statement above such statement block with the indent index $i - 1$ is constructed as their parent node. Secondly, we encode each node (i.e., each statement) of the tree into a vector by transformer. At last, we build the tree-transformer accordingly to further encode all the vector nodes of the tree for obtaining the code snippet representations. Specifically, we traverse the tree in a post-order manner. Assuming a node $n_i$ and its parent node $n_j$, if $n_i$ is a leaf node, we replace the vector of $n_j$, namely $V_{n_j}$, by the vector list $\{V_{n_i}, V_{n_j}\}$ and subsequently traverse $n_j$’s other child nodes; otherwise, we traverse $n_i$’s child nodes. Next, we encode node $n_j$ with the updated vector list $\{V_{n_i}, V_{n_j}\}$ by transformer when it has
def DeepDependencyTargets(target_dicts, roots):
    dependencies = set()
pending = set(roots)
    while pending:
        r = pending.pop()
        return list(…)
        if (r in dependencies):
            continue

Figure 5: The Source Code and the Tree Structure.

no child nodes. The tree-transformer encoding process is shown as Algorithm 1.

Figure 5 illustrates indent-based tree representation of the code snippet given in Figure 2. We use this example to describe how the tree-transformer-based encoder works. Specifically in Figure 2, we construct nodes “Dependencies = set()” and “pending=set(roots)”, “while pending” and “return list(...)” as siblings because they are assigned with the same indents and one-shorter-indent preceding statement “def DeepDependencyTargets(target_dicts, roots):”, which is constructed as their parent node. Then, we encode all the statement nodes into vectors by transformer respectively. Next, as the root’s child nodes “Dependencies = set()” and “pending=set(roots)” are leaf nodes, we replace the root vector by the vector list of them three. Then, since the root’s child node “while pending” is not the leaf node, we first encode its child node “if (r in dependencies):” with “continue” by transformer, and then encode the resulting vector with the siblings of “while pending” and “if (r in dependencies)” together by transformer. At last, we encode the root node with all its child nodes to obtain the final representation of this code snippet.

4.2 Code Summarization

Initially by collecting code snippets with their associated comments and forming <code; comment > pairs for training the code summarization model, the code summarization component is implemented via reinforcement learning (i.e., the actor-critic framework), where the actor network establishes an encoder-decoder mechanism to derive code comments and the critic network iteratively provides feedback for tuning the actor network. In particular, the actor network leverages a transformer-based or a tree-transformer-based encoder to encode the collected code into hidden space vectors and applies a transformer-based decoder to decode them to natural language comments. Next, by computing the similarity between the generated and the ground-truth comments, the critic network iteratively provides feedback for tuning the actor network. As a result, given a code snippet, its corresponding natural language comment can be generated based on the trained code summarization model.

4.2.1 Actor Network. The actor network is composed of an encoder and decoder.

4.2.2 Critic Network. To enhance the accuracy of the generated code comments, Trans3 applies a critic network to approximate the value of the generated comments at time t to issue a feedback to tune the network iteratively. Unlike the actor network which outputs a probability distribution, the critic network outputs a single value 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,

$$V(s_t) = \mathbb{E}_{y_{t+1:T} \mid y_{0:T}} \left[ \sum_{t=0}^{T-t} r_{t+1} \mid y_{t+1}, \cdots, y_T, h \right]$$

where $T$ is the max step of decoding and $h$ 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 $T$ or generates the end-of-sequence (EOS) token. For instance, a BLEU-based reward function can be calculated as:

$$r = \exp \left( \frac{1}{N} \sum_{i=1}^{N} \log p_n \right)$$

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

4.2.3 Model Training. For the actor network, the training objective is to minimize the negative expected reward, which is defined as $\mathcal{L}(\theta) = -\mathbb{E}_{y_{0:T} \sim \pi} \left( \sum_{t} r_t \right)$, where $\theta$ is the parameter set 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.
We conduct a set of extensive experiments on the effectiveness of our proposed approach, and use the generated comments into vector sets by tree-transformer-based encoder and transformer-based encoder respectively, and encodes the query into a vector by transformer-based encoder. Next, we compute the similarity scores between the query vector and the vectors in both the code snippets vector set and the generated comments vector set. At last, we rank all the code snippets to recommend the search results derived from the linear combination of the two similarity score sets which are trained for optimality.

As shown in Figure 3 (b), we encode the code snippets and the generated comments into vector spaces \( \{V_c\} \) and \( \{V_s\} \) by the tree-transformer-based encoder and transformer-based encoder respectively. We also encode the given natural language query into a vector \( V_q \) by transformer-based encoder. Next, we compute the similarity scores between \( V_q \) and the vectors of the code snippet \( i \) from both \( \{V_c\} \) and \( \{V_s\} \) as \( \text{sim}(V_q, V_{ci}) \) and \( \text{sim}(V_q, V_{si}) \). Furthermore, we derive the weighted score of the code snippet \( i \), \( \text{score}_i \), by linearly combining \( \text{sim}(V_q, V_{ci}) \) and \( \text{sim}(V_q, V_{si}) \), as shown in Equation 10. Eventually, we rank all the code snippets according to their \( \text{scores} \) for recommending the search results,

\[
\text{score}(Q, C) = \beta \times \text{sim}(V_q, V_{ci}) + (1 - \beta) \times \text{sim}(V_q, V_{si})
\]  

where \( \beta \) is a parameter that ranges from 0 to 1 and determined after training, \( \text{sim}() \) is computed by cosine. Specially, given the query \( q_i \), the training objective is to ensure \( \text{score}(q_i, c_i) > \text{score}(q_i, c_j) \), where \( c_j \) demonstrates the code snippets in the dataset expect \( c_i \).

5 EVALUATION

We conduct a set of extensive experiments on the effectiveness and efficiency of \( \text{TranS}^3 \) in terms of both the code summarization and code search components compared with state-of-the-art approaches.

5.1 Experimental Setups

To evaluate the performance of our proposed approach, we use the dataset presented in [37] where over 120,000 <code:comment> pairs are collected from various Python projects in GitHub [53] with 50,400 code tokens and 31,350 comment tokens in its vocabulary respectively. For cross validation, we shuffle the original dataset and use the first 60% for training, 20% for validation, and the rest for testing.

In our experiments, the word embedding size is set to 1280, the batch size is set to 2048, the layer size is set to 6, and the head number is set to 8. We pretrain both actor network and critic network with 20000 steps each, and train the actor-critic network with 100000 steps simultaneously. For the code search part, the comments in the dataset are utilized for the query. All the experiments in this paper are implemented with Python 3.5, and run on a computer with a 2.8 GHz Intel Core i7 CPU, 64 GB 1600 MHz DDR3 RAM, and a Saturn XT GPU with 24 GB memory running RHEL 7.5.

5.2 Result Analysis

5.2.1 Code summarization. To evaluate the code summarization component of \( \text{TranS}^3 \), we select several state-of-the-art approaches, i.e., Hybrid-DeepCom [54], CoaCor [6], and AutoSum [55] for performance comparison with \( \text{TranS}^3 \). In particular, Hybrid-DeepCom [54] utilizes the AST sequence converted by traversing the AST as the code representation and input the AST sequence to the GRU-based NMT for code summarization via combining lexical and structural information. CoaCor [6] utilizes the plain text of source code and an LSTM-based encoder-decoder framework for code summarization. AutoSum [55] utilizes a tree-LSTM-based NMT model and inputs the code snippet as plain text to the code-generation model with reinforcement learning for performance enhancement. Similarly, the evaluation for \( \text{TranS}^3 \) is also designed to explore the performance of its different components, where \( \text{TranS}^3_{\text{base}} \) adopts the transformer-based encoder; \( \text{TranS}^3_{\text{tree}} \) adopts the tree-transformer-based encoder for source code; and \( \text{TranS}^3_{\text{tree+RL}} \), i.e., the complete \( \text{TranS}^3 \), utilizes the tree-transformer-based encoder for source code and reinforcement learning for further enhancing the code summarization model.

We evaluate the performance of all the approaches based on four widely-used evaluation metrics adopted in neural machine translation and image captioning: BLEU [56], METEOR [57], ROUGE [58] and CIDER [59]. In particular, BLEU measures the average n-gram precision on a set of reference sentences with a penalty for short sentences. METEOR evaluates how well the results capture content from the references via recall which is computed via stemming and synonym matching. ROUGE-L imports account sentence level structure similarity and identifies the longest co-occurrence in sequential n-grams. CIDER is a consensus-based evaluation protocol for image captioning that evaluates the similarity of the generated comments and the ground truth.

Table 1 demonstrates the code summarization results of all the approaches in terms of the selected metrics. While the compared approaches achieve close performances, e.g., around 20% in terms of BLEU-1, \( \text{TranS}^3 \) can approximate 38%. In particular, we can obtain the following detailed findings. First, we can observe that \( \text{TranS}^3 \) can significantly outperform all the compared approaches in terms of all the evaluated metrics. For instance, the complete \( \text{TranS}^3 \), i.e., \( \text{TranS}^3_{\text{tree+RL}} \) can outperform all the compared approaches from 47.2% to 141.6% in terms of BLEU-1. Such performance advantages can indicate the superiority of the transformer-enabled self-attention mechanism over the mechanisms, including the attention mechanism, that are adopted in other RNN-based approaches, because the self-attention mechanism can effectively capture the...
impacts of the overall text on all the tokens of the input sequences for better reflecting their semantics and thus optimizing the language model weights. Next, we can verify that each component of TranS$^3$ is effective for enhancing the performance. For instance, by applying the tree-transformer-based encoder, TranS$^3$$_{tree}$ can dramatically outperform TranS$^3$$_{base}$ that only applies the transformer-based encoder by 15.7% in terms of BLEU-1. We can verify that our tree transformer based on identifying and leveraging the indent-based program semantic granularity can effectively strengthen the language model by augmenting the semantic level information for tokens. Moreover, by applying reinforcement learning, TranS$^3$$_{tree+RL}$ outperforms TranS$^3$, i.e., TranS$^3$$_{tree}$ by 17.5% in terms of BLEU-1, which can further verify the strength of reinforcement learning as verified in [6, 55]. Note that the performance of certain approaches, e.g., Hybrid-DeepCom, AutoSum, and TranS$^3$ which all utilize the generated comments to strengthen their code search performance, their adopted code summarization metrics are essentially based on word frequency which generally are fine-grained, e.g., BLEU-based metrics, while their code search metrics are generally based on coarse-grained query-wise comparisons. Therefore, the code summarization metrics tend to result in distinguishable results for different techniques because they are likely to reflect the trivial difference between two generated comments. However, their corresponding code search results might not be that distinguishable because the two generated comments might be trained to result in the result in the identical code rankings. For instance, suppose two code summarization approaches generate the comments “returns the path of the target dependencies” and “derive a target-dependency list” respectively. While they can be used to represent the same code snippets, they may result in different BLEU scores because they consist of different words. However, if they are used for code search, they can both rank the code snippet of Figure 2 on the top and thus result in the identical score in terms of the code search metrics. (2) CoaCor [6] can approach a close performance to TranS$^3$ because its rewarding mechanism utilizes the search accuracy to guide the code annotation generation and search modeling directly. However, we can observe that TranS$^3$ significantly outperforms CoaCor in terms of code summarization (by 47.2%). Therefore, to bridge such performance gap, CoaCor has to pay extra effort for enhancing its modeling process while TranS$^3$ can limit its effort in training the model once and for all for optimizing both code summarization and code search.

We also conduct a case study to evaluate the effectiveness of TranS$^3$. We organized 5 postgraduate students and 5 developers with certain Python background. We designed 15 programming tasks where each participant is asked to choose 3 tasks for code search using TranS$^3$ as well as our benchmark. Two example tasks are listed as follows:
Table 2: Sample issues for code summarization case study

| Issue link | Generated comment | Feedback |
|------------|-------------------|----------|
| 1 | https://github.com/mikunit567/GAE/issues/1 | Validate a given xsrf token by retrieving it. | “Yes, this is correct. Validate a retrieved XSRF from the memory cache and then with the token perform an associated action.” |
| 2 | https://github.com/hamzafaisaljarral/scoop/issues/1 | Iterates through the glob nodes. | “Yup you have got that right but for better understanding you have to look into django-shop documentation and look into django-cms documentation as well.” |
| 3 | https://github.com/rumd3x/PSP-POC/issues/1 | Combine two lists in a list. | “The pstats package is used for creating reports from data generated by the Profiles class. The add_callers function is supposed to take a source list, and a target list, and return new callers by combining the call results of both target and source by adding the call time.” |

Table 3: Code search accuracy compared with baselines. (Best scores are in boldface.)

| Approaches  | MRR   | nDCG  | SR@5 | SR@10 |
|-------------|-------|-------|------|-------|
| DeepCS      | 48.41 | 58.85 | 57.44| 66.78 |
| CoaCor      | 59.33 | 67.51 | 67.05| 73.58 |
| Hybrid-DeepCom | 50.92 | 59.92 | 60.52| 68.35 |
| AutoSum     | 57.68 | 63.52 | 63.43| 70.16 |
| Trans\textsuperscript{3}\textsubscript{base} | 58.43 | 65.13 | 63.28| 70.85 |
| Trans\textsuperscript{3}\textsubscript{tree} | 60.57 | 68.43 | 65.16| 74.13 |
| Trans\textsuperscript{3}\textsubscript{tree+RL} | 62.37 | 70.62 | 66.95| 75.21 |

- Task 1: Remove all the files in a directory.
- Task 2: Sends a message to the admins.

Then, they are asked to evaluate if the searched code snippets can solve the tasks or are helpful for solving them, by giving a score on a five-point Likert scale (strongly agree is 5 and strongly disagree is 1). For the 10 participants, the average Likert score is 3.167 (with standard deviation of 1.472), which indicates that in general, the efficacy of Trans\textsuperscript{3} can be acceptable.

6 Threats to validity

There are several threats to validity of our proposed approach and its results, which are presented as follows.

The main threat to internal validity is the potential defects in the implementation of our techniques. To reduce such threat, we adopted a commonly-used benchmark with over 120,000 Python functions for evaluating the effectiveness and efficiency of our proposed approach and several existing approaches for comparison. Moreover, to ensure the fair comparison, we directly downloaded the optimized models of the existing approaches for comparison.

The threats to external validity mainly lie in the dataset quality and the evaluation metrics of our experiments. On one hand, the quality of the training data, i.e., the $<\text{code;}\text{comment}>$ pairs adopted in our experiment was not evaluated. Among the over 120,000 python functions, it is likely that part of the poor-quality data can taint the training results. However, since (1) all the approaches were evaluated in the identical benchmark, and (2) the adopted evaluation metrics measure the performance of the approaches by word frequency where the corresponding performance difference among the approaches can indicate their word mapping levels, we can also infer that given high-quality training data, the performance distribution of all the approaches are likely to maintain consistency, where Trans\textsuperscript{3} can still outperform the other approaches in terms of the word-frequency metrics. Moreover, the performance of the tree-transformer-based encoder heavily relies on the quality of program forms. However, the experimental results indicate that the tree-transformer-based encoder can achieve better performance than the transformer-based encoder regardless the quality of the program forms. On the other hand, the word-frequency-based metrics cannot fully reflect the semantic correctness of the approaches. To reduce such threat, we adopted a set of empirical studies such that developers can feedback for the quality of our code summarization and code search results. The positive study results can strengthen the validity of the effectiveness and efficiency of Trans\textsuperscript{3}.

7 Related Work

7.1 Code Summarization

The code summarization techniques can be mainly categorized as information-retrieval-based approaches and deep-learning-based approaches.

Information-retrieval-based approaches. Wong et al. [26] proposed AutoComment which leverages code-description mappings to generate description comments for similar code segments matched in open-source projects. Similarly they also apply code clone detection techniques to find similar code snippets and extract comments from the similar code snippets [63]. Movshovitz-Attias et al. [2] predicted comments from Java source files using topic models and n-grams. Haiduc et al. [64] combined IR techniques, i.e., Vector Space Model and Latent Semantic Indexing, to generate terms-based summaries for Java classes and methods.
7.2 Code Search

Code search techniques also mainly consists of information-retrieval-based approaches and deep-learning-based approaches.

**Information-retrieval-based approaches.** Hill et al. [67] proposed CONQUER which integrates multiple feedback mechanisms into the search results view. Some approaches proposed to extend the queries, for example, Lu et al. [27] proposed to extend queries with synonyms generated from WordNet and then match them with phrases extracting from code identifiers to obtain the search results. Lv et al. [4] designed a API understanding component to figure out the potential APIs and then expand the query with the potential APIs and retrieve relevant code snippets from the codebase. Similarly, Raghothaman et al. [68] proposed swim, which first suggests an API set given a query by the natural language to API mapper that is extracted from clickthrough data in search engine, and then generates code using the suggested APIs by the synthesizer.

**Deep-learning-based approaches** The deep learning-based approaches usually encode the code snippets and natural language query into a hidden vector space, and then train a model to make the corresponding code and query vector more similar in the hidden space. Gu et al. [5] proposed DeepCS, which reads code snippets and embeds them into vectors. Then, given a query, it returns the code snippets with the nearest vectors to the query. Luan et al. [69] proposed Aroma, which takes a code snippet as input, assembles a list of method bodies that contain the snippet, clusters and intersects those method bodies to offer code recommendations. Different from the above approaches, Akbar et al. [30] presented a framework that incorporates both ordering and semantic relationships between the terms and builds one-hot encoding model to rank the retrieval results. Chen et al. [29] proposed BVAE, which includes C-VAE and L-VAE to encode code and query respectively, based on which semantic vector for both code and description and generate completely. Yao et al. [6] proposed CaoCor, which designs a rewarding mechanism to guide the code annotation model directly based on how effectively the generated annotation distinguishes the code snippet for code retrieval.

**Other approaches.** Sivaraman et al. [70] proposed ALICE, which integrates active learning and inductive logic programming to incorporate partial user feedback and refine code search patterns. Takuya et al. [71] proposed Selene, which uses the entire editing code as query and recommends code based on a associative search engine. Lemons et al. [72] proposed a test-driven code search and reuse approach, which searches code according to the behavior of the desired feature to be searched.

8 CONCLUSION

In this paper, we propose \textit{Trans$^3$}, which is a transformer-based framework to integrate code summarization with code search. Specifically, \textit{Trans$^3$} enables an actor-critic network for code summarization. In the actor network, we build transformer- and tree-transformer-based encoder to encode code snippets and decode the given code snippet to generate their comments. Meanwhile, we utilize the feedback from the critic network to iteratively tune the actor network for enhancing the quality of the generated comments. Furthermore, we import the generated comments to code search for enhancing its accuracy. We conduct a set of experimental studies and case studies to evaluate the effectiveness of \textit{Trans$^3$}, where the experimental results suggest that \textit{Trans$^3$} can significantly outperform multiple state-of-the-art approaches in both code summarization and code search and the study results further strengthen the efficacy of \textit{Trans$^3$} from the developers’ points of view.

REFERENCES

[1] A. LeClair, S. Jiang, and C. McMillan, “A neural model for generating natural language summaries of program subroutines,” in Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019, pp. 795–806.
[2] 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 (Volume 2: Short Papers), vol. 2, 2013, pp. 35–40.
[3] G. Sridhara, E. Hill, D. Mupparani, 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.
[4] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, “Codehew: Effective code search based on api understanding and extended boolean model (e),” in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015, pp. 260–270.
[5] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 933–944.
[6] Z. Yao, J. R. Peditmaul, and H. Sun, “Cao Cor: Code annotation for code retrieval with reinforcement learning,” in The World Wide Web Conference. ACM, 2019, pp. 2203–2214.
[7] B. P. Lienart and E. B. Swanson, “Software maintenance management,” IEEE Proceedings E Computers and Digital Techniques Transactions on Software Engineering, vol. 127, no. 6, 1980.
[8] L. Moreno and A. Marcus, “Automatic software summarization: the state of the art,” in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017, pp. 511–512.
[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, pp. 132–142. [Online]. Available: https://doi.org/10.1145/3238147.3238187.
