CCMC: Code Completion with a Memory Mechanism and a Copy Mechanism

Hao Yang
School of Computer Science and Engineering
Central South University
Changsha, China
cspikachuhy@csu.edu.cn

Li Kuang*
School of Computer Science and Engineering
Central South University
Changsha, China
kuangli@csu.edu.cn

ABSTRACT

Code completion tools are increasingly important when developing modern software. Recently, statistical language modeling techniques have achieved great success in the code completion task. However, two major issues with these techniques severely affect the performance of neural language models (NLMs) of code completion. a) Long-range dependences are common in program source code. b) New and rare vocabulary in code is much higher than natural language. To address the challenges above, in this paper, we propose code completion with a memory mechanism and a copy mechanism (CCMC). To capture the long-range dependencies in the program source code, we employ Transformer-XL as our base model. To utilize the locally repeated terms in program source code, we apply the pointer network into our base model and design CopyMask to improve the training efficiency, which is inspired by masked multihead attention in the transformer decoder. To combine the long-range dependency modeling ability from Transformer-XL and the ability to copy the input token to output from the pointer network, we design a memory mechanism and a copy mechanism. Through our memory mechanism, our model can uniformly manage the context used by Transformer-XL and pointer network. Through our copy mechanism, our model can either generate a within-vocabulary token or copy an out-of-vocabulary (OOV) token from inputs. Experiments on a real-world dataset demonstrate the effectiveness of our CCMC on the code completion task.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Code completion, Transformer, Pointer network, Memory

*Li Kuang is the corresponding author.

ACM Reference Format:
Hao Yang and Li Kuang. 2021. CCMC: Code Completion with a Memory Mechanism and a Copy Mechanism. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3463274.3463332

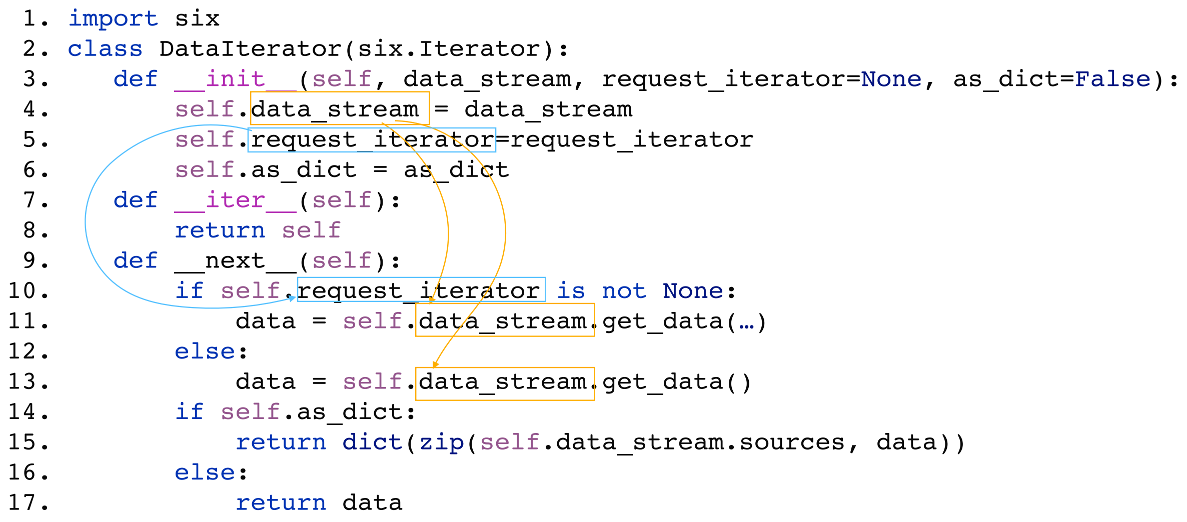
1 INTRODUCTION

When developing software, integrated development environments (IDEs) provide a series of helpful services to improve the development process, especially its powerful intelligent code completion. The IDE offers a drop-down list for software developers to choose the most likely next tokens (e.g., right libraries, APIs, variables), which are based on the code already written and cursor position on the screen. However, traditional intelligent code completion approaches rely on either compile-time information (e.g., variable type) or term frequency statistics and heuristics rules [11], which limit their ability in dynamically typed languages (e.g., Python) and cannot capture software developers' programming patterns well.

To alleviate this problem, researchers have turned to learning-based language models in recent years[4, 22, 27]. They treat programming languages as natural languages and train code completion models with large code corpora since programming language is a specific language for communicating with machines. N-gram is one of the most widely used language models in code completion[12, 13, 25]. The recurrent neural network (RNN), in particular, the long short-term memory (LSTM) network [14], is another language model that has obtained good results on multiple code completion benchmarks[3, 18, 19] and hence, is a standard solution to the language model.

However, these standard neural language models are limited by their long-range dependency ability. Previous work has found that LSTM language models use 200 context words on average [15], which degrades their performance on code completion tasks because long-range dependencies are very pervasive in program source code. For example, developers often declare a variable first and use it after many lines of source code. As can be seen in Figure 1, a variable named `data_stream` is declared in function `__init__` in line 4 and is used in function `__next__` in line 11, line 13 and line 15. Between function `__init__` and function `__next__`, there are other functions defined (e.g., function `__iter__` in our example code snippet), which make the long-range dependencies more complex.

Another critical issue in code completion is the out-of-vocabulary (OOV) problem. On the one hand, the code completion task has a very large vocabulary since developers usually create variables according to their preference. Considering computational efficiency, we cannot build a fairly large vocabulary in which most words are



```
1.  import six
2.  class DataIterator(six.Iterator):
3.      def __init__(self, data_stream, request_iterator=None, as_dict=False):
4.          self.data_stream = data_stream
5.          self.request_iterator=request_iterator
6.          self.as_dict = as_dict
7.      def __iter__(self):
8.          return self
9.      def __next__(self):
10.         if self.request_iterator is not None:
11.             data = self.data_stream.get_data(…)
12.         else:
13.             data = self.data_stream.get_data()
14.         if self.as_dict:
15.             return dict(zip(self.data_stream.sources, data))
16.         else:
17.             return data
```

Figure 1: An example of the long-term dependency and out-of-vocabulary (OOV) problem. The blue line and arrow indicate the oov word, and the orange line and arrow indicate the vocabulary word.

used at low frequencies. On the other hand, developers still create new vocabulary in source code, which makes models with large vocabularies inefficient. Fortunately, developers often create a new word and use it later. For example, in Figure 1, `request_iterator`, an out-of-vocabulary word, is created in function `__init__` in line 5 and used in function `__next__` in line 10. This locality makes it possible for us to copy previous out-of-vocabulary words from the input source code sequence to the current prediction position.

To address the challenges above, in this article, we propose **C**ode **C**ompletion with **M**emory mechanism and **C**opy mechanism (CCMC), a transformer-based code completion model. We employ Transformer-XL, a variant of Transformer, as our base model to capture the long-range dependency in the program source code. To predict the OOV token, we enhance our base model by integrating a pointer network. We design a memory mechanism and a copy mechanism to combine the long-range dependency modeling ability from Transformer-XL and the ability to copy the input token to output from the pointer network. To unify the management of the context used by Transformer-XL and pointer network, the memory mechanism also cached previous inputs and hidden states, which are useful for our copy mechanism. Through our copy mechanism, our model can either generate a within-vocabulary token or copy an OOV token from inputs. Inspired by masked multihead attention in the transformer decoder, we also design CopyMask to improve training efficiency when applying the pointer network to our transformer-based code completion model. We evaluate the performance of our model on a real benchmarked dataset (PY150). The results show the effectiveness of our CCMC on the code completion task. The main contributions of this work are as follows:

- We propose a Transformer-based code completion model called CCMC to model the long-range dependency on the code completion task.
- We design a memory mechanism and a copy mechanism for our Transformer-based code completion model to predict the OOV token and design CopyMask to improve training efficiency.
- Our proposed model achieves better results compared to previous RNN-based state-of-the-art code completion model on a real-world dataset.

The remainder of this paper is organized as follows. Section 2 discusses some work related to this paper. Section 3 provides background knowledge on the language model and self-attention. Then, we introduce our proposed model in Section 4. Section 5 shows the experiments and evaluation. Finally, we conclude our paper in Section 6.

2 RELATED WORK

2.1 Language Model in Code Completion

Much of the work is inspired by Hindle et al. [13], who demonstrated that the regularity of "natural" software can be captured by a statistical model. Recurrent neural networks (RNNs), in particular, long short-term memory (LSTM) networks, have obtained strong results on multiple code completion benchmarks and are widely used in code completion models. Raychev et al. [22] explore how to use recurrent neural networks (RNNs) on the code completion task. White et al. [27] demonstrated that a relatively simple RNN configuration can outperform n-grams and cache-based n-grams on a Java corpus. However, RNNs suffer from the gradients vanishing/exploding problem. To mitigate this problem, a variant of RNNs (LSTM) is proposed. Liu et al. [19] explore how to use long short-term memory (LSTM) networks to automatically learn code completion from a large corpus of dynamically typed JavaScript code. To alleviate the hidden state bottleneck of LSTM, Li et al. [18] proposed an attention-enhanced LSTM by keeping an external memory of previous hidden states and achieved state-of-the-art results in the PY150 dataset, which we also used in our work. Transformers [26] are sequence-to-sequence neural networks based solely on attention mechanisms, which have achieved state-of-the-art results [7, 8, 20] in many NLP tasks, including question answering, sentence entailment and language modeling. Recently, Transformers have become increasingly popular for code completion, as they can capture longer dependencies than RNN-based models. Kim et al. [16] explored how to feed an abstract syntax tree (AST) to Transformers. They demonstrated that exposing the syntactic structure of code is a practical way to obtain better accuracy from Transformers. However, they do not consider the out-of-vocabulary(OOV) problem. There have also been reported applications of Transformer-based code completion models such as Galois and TabNine.

2.2 Pointer Network

The pointer network [24] is a sequence-to-sequence model that produces an output sequence consisting of words from the input sequence. The pointer network is shown to be helpful in language modeling, especially in solving the OOV problem[9, 10, 23]. See et al. [23] propose a pointer-generator network, which allows both copying words by pointing and generating words from a fixed vocabulary. Bhoopchand et al. [3] propose a sparse pointer network to predict Python identifiers. Li et al. [18] propose a pointer mixture network that learns to either copy OOV values from the input sequence or generate new values from the RNN output.

3 PRELIMINARIES

In this section, we briefly introduce the preliminaries of our CCMC model: the neural language model and attention.

3.1 Neural Language Model

The task of code completion can be approached by a language model that estimates the probability of observing sequences of tokens in Python programs. For example, for the sequence $S = w_1, \ldots, w_n$, the joint probability of S factorizes according to

$$P_\theta(S) = P_\theta(w_1) \cdot \prod_{t=2}^{n} P_\theta(w_t \mid w_{t-1}, \ldots, w_1) \tag{1}$$

where w_t is the t-th token in the sequence S, and the parameters θ are estimated from a training corpus.

Given a sequence of Python tokens, we seek to predict the next tokens w_{n+1} that maximize

$$\arg\max_{w_{n+1}} P_\theta(w_1, \ldots, w_n, w_{n+1}) \tag{2}$$

In this work, we build upon neural language models using transformer. This neural language model estimates the properties in (1) using the output vector of the transformer at time step t (denoted h_t here) according to

$$P_\theta(w_t = \tau \mid w_{t-1}, \ldots, w_1) = \frac{\exp\left(v_\tau^T h_t + b_\tau\right)}{\sum_{\tau'} \exp\left(v_{\tau'}^T h_t + b_{\tau'}\right)} \tag{3}$$

where w_t is the t-th token in the sequence S, h_t is the hidden states in time step t, v_τ is a parameter vector associated with token τ in the vocabulary, τ' is each time step in sequence S, $v_{\tau'}$ is its hidden state vector, b_τ is the bias, and exp denotes the exponential operation.

3.2 Transformer

The Transformer is a deep neural network that is based solely on attention mechanisms. In general, the Transformer can be seen as a stack of many transformer layers.

$$\text{Transformer-Layer}(Q, K, V) = \text{FFN}(\text{Attn}(Q, K, V)) \tag{4}$$

Each transformer layer consists of self-attention and positionwise feed-forward networks. In our model, we use the Transformer-Layer to refer to calculations in each transformer layer. The other calculations detail can be found in [26].

Self-attention is the mechanism used by the transformer, which applies the attention function to each word in the sequences. 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[26]. The output is the weighted sum of the values, where the weight is computed by a function of the query with the corresponding key. The calculation of attention is shown in equation 5.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{5}$$

where Q, K, V are the query, key and value vectors, respectively, d_k is the dimension of the query vector, and $.^T$ represents the transpose operation.

The transformer also uses multihead attention, which allows the model to focus on information from different representation subspaces from different locations. In each head, the calculation of attention (6) is the same as equation (5).

$$O_i^{head} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{6}$$

where i is the index of the head, $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W_i^O \in \mathbb{R}^{hd_k \times d_{model}}$. The output of each attention is concatenated and then projected to model dimension d_k, shown as (7).

$$\text{MultiHead}(Q, K, V) = \text{Concat}(O_1^{head}, ..., O_n^{head})W^O \tag{7}$$

where $W^O \in \mathbb{R}^{nd_k \times d_{model}}$ is a learnable parameter.

The positionwise feed-forward network is another component in the transformer layer, which is a fully connected feed-forward network consisting of two linear layers with a ReLU activation function. Formally, the positionwise feed-forward network can be described as:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{8}$$

where W_1, W_2, b_1, b_2 are learnable parameters.

4 APPROACH

In this section, we present our approach for the next-token code completion task. We first present an overview of the network architecture of our proposed model, and then we present the program representation in our model. Finally, we introduce each component of the model in detail.

4.1 Overall Architecture

Figure 2 shows the architecture of our proposed model. We employ Transformer-XL, a variant of the Transformer, as our base model, not only because it can model very long dependencies in the natural language process but also, more importantly, because its recurrent mechanism is helpful for our copy mechanism. The AST node sequences of the program are fed into the transformer layer, producing a sequence of hidden states h_i. On each step t, we calculate a copy probability $p_{copy} \in [0, 1]$ by copy attention context vector c_i and hidden status h_i, then we obtain the final distribution y_t by weighted summing the value distribution w_t and the copy distribution l_t. Our memory mechanism cached previous AST node sequences and hidden states to provide extended context for the transformer layer and copy mechanism. Through our copy mechanism, our model can either generate a within-vocabulary word or copy an OOV word from previous input tokens. To train our model with copy effectively, we design CopyMask to make our copy attention calculation parallel, which is inspired by masked multihead attention in the transformer decoder. We also use joint training for type prediction and value prediction since type and value are relevant in AST.

4.2 Program Representation

In our dataset, each program is represented in the form of an abstract syntax tree (AST). Programming language has an unambiguous context-free grammar (CFG), so each program can be parsed into a unique AST through a compiler. Representing programs as ASTs rather than plain text enables us to obtain the structure of the program. We serialize each AST as a sequence of nodes in the in-order depth-first traversal, as transformers are sequence models and cannot handle tree data structures directly.

As can be seen from Figure 3, a Python program code snippet `np.array(arr)` is parsed as AST. Each AST node has two properties: one is the node type, and the other is the optional value. Only leaf nodes in AST have value property. In this example, `Expr`, `Call`, `AttributeLoad` have no value properties. To process the type and value properties consistently, we use a special token EMPTY as its value for nonleaf nodes.

If we represent an AST node as $w_i = (type_i, value_i)$, where T_i is the node type and V_i is the value, then each program can be denoted as a sequence of words $w_{i=1}^{n}$. When feeding AST nodes to the Transformer, we train an embedding vector for type and value and then concatenate the two embeddings into one vector.

$$e^{node} = [e^{type} \circ e^{value}] \tag{9}$$

where $e^{node} \in \mathbb{R}^k, k = d_{type} + d_{value}$, $e^{type} \in \mathbb{R}^{d_{type}}$, and $e^{value} \in \mathbb{R}^{d_{value}}$, d_{type}, d_{value} is the embedding size of the type and value, respectively, and $\circ$ denotes the concatenation operation.

4.3 Base Model

Recently, Transformers have achieved appealing success in language modeling. However, vanilla Transformers suffer from the fixed-length context problem. Transformer-XL [6] is proposed to mitigate this problem by integrating a segment-level recurrence mechanism for transformer layers. Transformer-XL has been proven to outperform vanilla Transformer on various tasks, including language modeling. Therefore, we choose Transformer-XL as our base model for code completion.

The core of Transformer-XL is the segment-level recurrence mechanism, which allows Transformer-XL to exploit information in the history, leading to a longer range modeling ability. Formally, let the two consecutive segments of length L be $\mathbf{s}_\tau = [x_{\tau,1}, \ldots, x_{\tau,L}]$ and $\mathbf{s}_{\tau+1} = [x_{\tau+1,1}, \ldots, x_{\tau+1,L}]$, respectively. Denoting the n-th layer hidden state sequence produced for the τ-th segment $\mathbf{s}_\tau$ by $\mathbf{h}_\tau^n \in \mathbb{R}^{L\times d}$, where d is the hidden dimension. Then, the n-th layer hidden state for segment $\mathbf{s}_{\tau+1}$ is produced (schematically) as follows: An N-layer Transformer-XL with a single attention head can be summarized as follows. For $n = 1, ..., N$:

$$\widetilde{\mathbf{h}}_{\tau+1}^{n-1} = \left[\mathrm{SG}\left(\mathbf{h}_\tau^{n-1}\right) \circ \mathbf{h}_{\tau+1}^{n-1}\right], \tag{10}$$

$$\mathbf{q}_{\tau+1}^n, \mathbf{k}_{\tau+1}^n, \mathbf{v}_{\tau+1}^n = \mathbf{h}_{\tau+1}^{n-1}\mathbf{W}_q^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_k^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_v^\top, \tag{11}$$

$$\mathbf{h}_{\tau+1}^n = \text{Transformer-Layer}\left(\mathbf{q}_{\tau+1}^n, \mathbf{k}_{\tau+1}^n, \mathbf{v}_{\tau+1}^n\right). \tag{12}$$

where $\mathbf{h}_\tau^n$ is the n-th transformer-layer hidden status at segment τ, the function $SG(\cdot)$ stands for stop gradient, the notation $[\mathbf{h_u} \circ \mathbf{h_v}]$ indicates the concatenation of two hidden sequences along the length dimension, $\mathbf{q}, \mathbf{k}, \mathbf{v}$ is the query, key and value vectors in self-attention and $\mathbf{W}.$ denotes model parameters. For Transformer-Layer, we described in 3.2.

After obtaining the last layer output h_t^N, we feed it to two linear layers to produce the final distribution P:

$$P = softmax(W_1(W_2 h_t^N + b_2) + b_1) \tag{13}$$

where W_1, W_2, b_1 and b_2 are learnable parameters.

During training, the loss for timestep t is the negative log likelihood of the target token w_t^* for that timestep

$$loss_t = -logP(w_t^*) \tag{14}$$

and the overall loss for the program is:

$$loss = \frac{1}{T}\sum_{t=0}^{T} loss_t \tag{15}$$

where T is the length of the program.

4.4 Memory Mechanism & Copy Mechanism

Both large vocabularies and OOV issues are prevalent in program source code, as programmers can create various new variable names, method names and class names according to their programming habits. Considering the computational efficiency, we only use the K most frequent tokens to build the token vocabulary and replace all OOV tokens with a special token UNK.

We first extend the recurrence mechanism in Transformer-XL to our memory mechanism. In addition to providing memory for Transformer-XL layers, the memory mechanism also needs to cache inputs and hidden states for the next copy attention calculation in

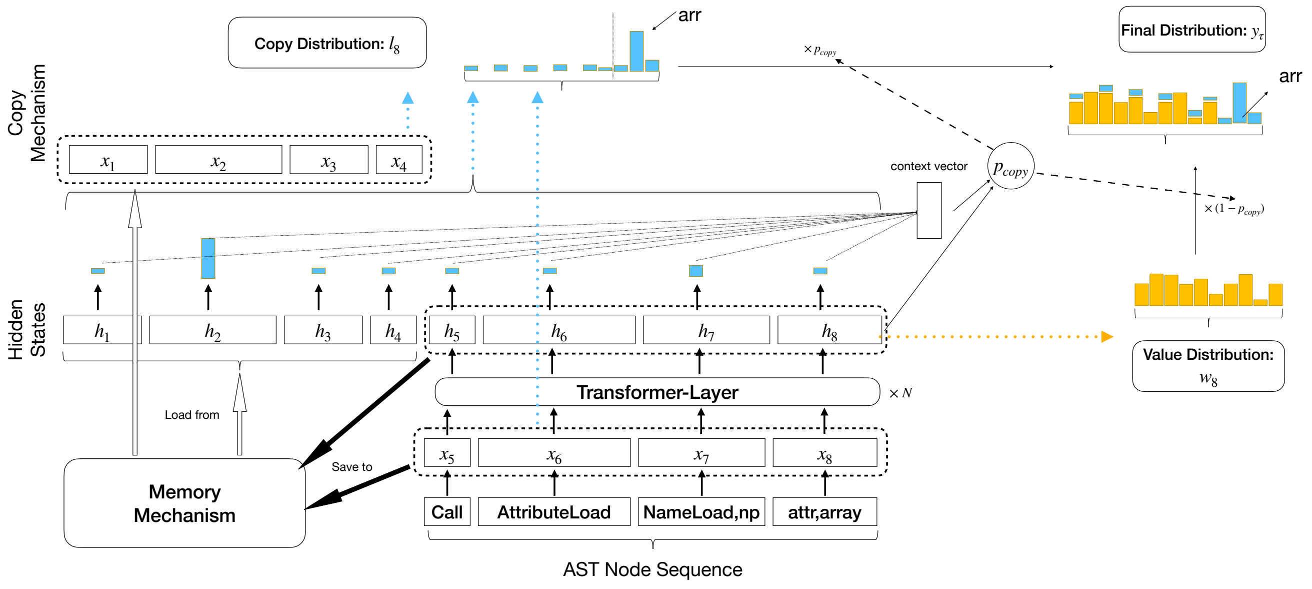


Figure 2: The CCMC model architecture. The memory cached pervious inputs x_1, x_2, x_3, x_4 and pervious transformer layer output hidden status h_1, h_2, h_3, h_4, which is used in calculating copy distribution l_τ and final distribution y_τ. For each timestep, a copy probability $p_{copy} \in [0, 1]$ is calculated by the copy attention context vector and transformer-layer output. Here, we ignore the p_{copy} calculation for simplicity. The value distribution w_τ and the copy distribution l_τ are weighted and summed to obtain the final distribution. Here, ⊕ indicates the elementwise addition operation.

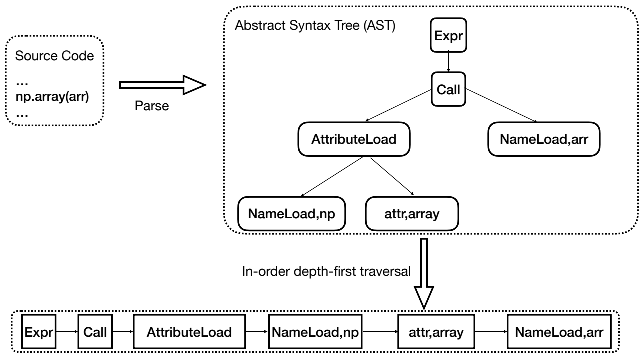


Figure 3: Program representation. A Python program code snippet is parsed as AST and then flattened to a sequence.

the copy mechanism. Formally, we keep an external memory of d_{mem} previous inputs and hidden statues:

$$M_t^w = [w_{t-d_{mem}}, ..., w_{t-1}] \tag{16}$$

$$M_t^h = [h_{t-d_{mem}}, ..., h_{t-1}] \tag{17}$$

Our copy mechanism builds upon the memory mechanism. The longest copy length is upper bounded by the memory size. We apply a pointer network to our base model. The pointer network points to local locations according to location weights. We design the copy mechanism as follows:

$$A_t = v^T tanh(W^m M_t^h + (W^h h_t) 1_L^T) \tag{18}$$

$$\alpha_t = softmax(A_t) \tag{19}$$

$$c_t = M_t^h \alpha_t^T \tag{20}$$

where $W^m, W^h \in \mathbb{R}^{k*k}$ and $v \in \mathbb{R}^k$ are trainable parameters. 1_L represents an L-dimensional vector of ones.

We use the location weights α_t as our copy distribution l_t according to our previous inputs from our memory mechanism, shown as (21). Note that if a token does not appear in the previous inputs, the copy probability of the token is zero; similarly, if the token appears in the previous inputs multiple times, we sum the probability of each occurrence.

$$l_t = \sum_i \alpha_t^i \ if \ w_i = w \tag{21}$$

where α_t^i is the weight of the i-th token w_i in previous inputs.

To combine the long-range dependency modeling ability from our base model and the ability to copy the input token to output from the pointer network, we first extend the global vocabulary distribution to our extended value vocabulary distribution w_t by setting the OOV distribution to 0. Then, we weighted the sum global vocabulary distribution w_t and local copy distribution l_t as our final distribution by a copy probability p_{copy}.

The copy probability $p_{copy} \in [0, 1]$ is calculated from the current hidden state h_t, current input e_t^{node} and copy context c_t:

$$p_{copy} = \sigma(W^p [h_t \circ e_t^{node} \circ c_t] + b^p) \tag{22}$$

where vector W^p and scalar b^p are learnable parameters, σ is the sigmoid function, and $\circ$ denotes the concatenation operation.

Next, p_{copy} is used as a soft switch to choose between generating a token from the vocabulary by sampling from w_t or copying a token from the input sequence by sampling from the local copy distribution l_t.

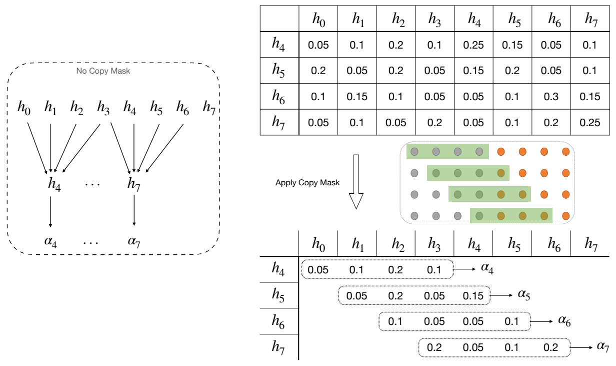
$$y_t = (1 - p_{copy})w_t + p_{copy}l_t \tag{23}$$

Finally, our model can both copy tokens from the previous input sequence and generate tokens from a fixed vocabulary.

4.5 Copy Mask

In each time step, we need to calculate location weights according to (18). Previous work [18] calculates each weight one by one, which seriously increases the training time. To address this issue, we design CopyMask to improve the training efficiency, which can calculate location weights in parallel, similar to the transformer decoder.

To better illustrate CopyMask, let us look at an example, see figure 4. Assume we obtained the current segment hidden states h_4, h_5, h_6, h_7 and obtained previous segment hidden states h_0, h_1, h_2, h_3 from our memory mechanism. If we follow the previous calculation method, we first calculate α_4 from $M_4^h = [h_0, ..., h_3]$ and h_4 according to (19) and update our memory $M_5^h = [h_1, ..., h_4]$ and h_5 according to (17). Then, we calculate $\alpha_5, \alpha_6, \alpha_7$ one by one, similar to calculating α_4. If we use CopyMask, we just need to perform a matrix multiplication operation and then mask the future information of the token. This is why our CopyMask is efficient.



	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7
h_4	0.05	0.1	0.2	0.1	0.25	0.15	0.05	0.1
h_5	0.2	0.05	0.2	0.05	0.15	0.2	0.05	0.1
h_6	0.1	0.15	0.1	0.05	0.05	0.1	0.3	0.15
h_7	0.05	0.1	0.05	0.2	0.05	0.1	0.2	0.25

	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7
h_4	0.05	0.1	0.2	0.1				
h_5		0.05	0.2	0.05	0.15			
h_6			0.1	0.05	0.05	0.1		
h_7				0.2	0.05	0.1	0.2	

Figure 4: Example of the copy mask.

4.6 Loss Function

To learn the type and value prediction tasks jointly, we sum the type prediction loss and value prediction loss as the final loss:

$$loss = loss_N + loss_T \tag{24}$$

where $loss_N$ is the loss of the next type of prediction and $loss_T$ is the loss of the next value prediction.

5 EVALUATION

5.1 Dataset

We evaluate our approach on a benchmarked dataset: Python (PY150), which contains 150,000 program files in AST formats, with 100,000 used for training and the remaining 50,000 used for testing. All of the programs in PY150 are collected from GitHub and publicly available. [1] This dataset has also been used in [18, 21]. As described in 4.2, all ASTs are serialized in an in-order depth-first traversal. The number of type properties and value properties of AST nodes and the queries (number of nodes) of the programs are shown in Table 1.

Table 1: Detail information of PY150

Type	Value
Type Vocabulary	181
Value Vocabulary	3.4×10^6
Training Queries	6.2×10^7
Test Queries	3.0×10^7

5.2 Data Processing

First, we traverse the ASTs and count the number of occurrences of each node type and value. Then, we build vocabularies for type and value. Since the number of types is only 181, there is no need to consider the OOV problem. For value, we use the K_{vocab} most frequent values in the training set to construct a global vocabulary of AST node values and mark all OOV node values in the training set and test set as unknown values. To meet the copy requirement, we assign a unique ID to each OOV token by keeping an extra dictionary for each program; thus, we build the extended vocabulary. More specifically, we traverse the ASTs again after constructing a global vocabulary. If a value appears in the global vocabulary, we use the ID in the global vocabulary. If the value is not in the vocabulary but in the extra dictionary, we use the ID in the extra dictionary. Otherwise, we add the value to the extra dictionary and assign a new ID as the value ID.

5.3 Metric

Similar to [18], we employ accuracy as our evaluation metric, which is the proportion of correctly predicted node type or value.

$$Accuracy = \frac{Number_{correct}}{Number_{total}} \tag{25}$$

If the target value is UNK, we treat all UNK predictions as incorrect predictions in both training and evaluation.

5.4 Experiment Setup

Our base model is a Transformer-XL network with a 6-layer transformer layer, memory size of d_{mem}, hidden unit size of 1,500, 5 heads and dimension of each head of 64. The embedding size of type d_{type} and value d_{value} are set to 300 and 1,200, respectively. To train the model, we use the cross-entropy loss function and Adam

[1] http://plml.ethz.ch

optimizer [17]. We set the initial learning rate as 0.00025 and clip the gradients' norm to 0.25. The batch size is 16, and we train our model for 8 epochs.

Like Li et al.[18]'s study, we divide each program into segments consisting of K_{seg} consecutive tokens, with the last segment being padded with EOF tokens if it is not full. The memory is initialized with m_0. We use old memory to calculate if both segments belong to the same program. Otherwise, the memory is reset to m_0. We initialize m_0 to be all-zero vectors, while other variables are randomly initialized using a normal distribution over the interval $[-0.02, 0.02]$.

We implement our models using TensorFlow [1] and run our experiments on Ubuntu 16.04 with NVIDIA GTX 2080 Ti. Unless otherwise stated, each experiment is run three times, and the average result is reported.

5.5 Research Questions

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

RQ1: Does our proposed CopyMask speed up training?

RQ2: Does a larger memory size help to achieve better accuracy?

RQ3: How does our proposed approach perform in OOV prediction?

RQ4: Do our proposed model provide better accuracy than state-of-the-art code completion model?

For RQ1, we compare the impact of the copy mask on the training time.

For RQ2, We compare the type prediction in different memory sizes d_{mem} from 50 to 800. For each experiment, we double our memory size to better observe the effect of the memory mechanism instead of increasing it by the same amount.

For RQ3, we compare value prediction in different OOV value rates. We change the global vocabulary size K_{vocab} for node values to be 1k, 10k and 50k to construct different OOV value rates.

For RQ4, we compare our model with the following RNN-based state-of-the-art baseline model and our base model.

- li et al.'s model [18]: an attention and pointer enhanced LSTM code completion model.
- vanilla Transformer-xl: a standard Transformer-xl network without any other components.

5.6 Results

Q1. For RQ1, we set $K_{seg} = 50, d_{mem} = 50, K_{vocab} = 50k$, and all other parameters are the same as 5.4. We compare the training time spent in each epoch.

Table 2: Training time spent in epoch epoch

	Training Time
CCMC (w/o copy mask)	4 h 1 m
CCMC (with copy mask)	3 h 7 m

As Table 2 shows, our model with CopyMask speeds 77.5% of the time compared to our model without a mask. We attribute this improvement to the fact that we can make full use of GPU matrix multiplication to accelerate. Therefore, the results demonstrate the effectiveness of our CopyMask.

Q2. For RQ2, we set d_{mem} from 50 to 800 and keep the other parameters the same as 5.4. Overall, we found that a larger memory size can further improve prediction accuracy but not infinitely.

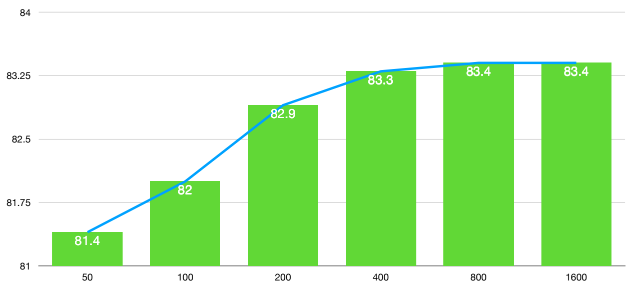


Figure 5: Type prediction accuracy with different memory sizes

As seen in Figure 5, our model with a large memory size outperforms the model with a small memory size on the next node's type prediction, achieving an improvement of 2.1%.

We attribute this improvement to the fact that with the memory size increment, our model can exploit more information in the historical context and model longer-range dependency. However, this improvement is not infinite. Note that when memory is already large, the improvement obtained by continuing to increase the memory size becomes very small, from 1% decreases to 0.1%, and the final approaches 0.

Q3. For RQ3, we K_{vocab} from $1k$, $10k$, $50k$ and keep the other parameters the same as 5.4.

Table 3: Accuracies on next value prediction with different vocabulary sizes

Vocabulary Size (OOV Rate)	1k	10k	50k
CCMC (w/o copy)	64.3%	69.6%	71.4%
CCMC (with copy)	65.2%	70.1%	71.6%

As Table 3 shows in each column, when the vocabulary size increases, our model obtains better accuracy. When applying the copy mechanism to our model, we can see improvements in value prediction of 0.9%, 0.5%, and 0.2%, respectively.

The reason lies in that 1) a larger vocabulary provides more information to our model, 2) when we add a copy mechanism to the model, we can select one most likely unknown token from an enhanced context consisting of memory and current hidden states, which gives our model a chance to choose the previous OOV value as the current prediction value and finally increase the accuracy. With increasing vocabulary size, the improvement in the copy mechanism becomes small. We attribute this to the fact that as the OOV ratio decreases, the unknown value that can be copied from the previous input sequence decreases, and finally, the effect of copying decreases. Therefore, the results demonstrate the effectiveness of our copy mechanism.

Q4. For RQ4, Table 4 lists the corresponding statistics and experimental results.

Table 4: Accuracies on next type and value prediction for PY15

	TYPE	VALUE
li et al.'s model	80.6%	70.1%
vanilla Transformer-xl	82.3%	69.8%
CCMC (ours)	83.4%	71.6%

As Table 4 shows, our model achieves the highest accuracy on both type prediction and value prediction, significantly improving the best records on the dataset PY150. When compared to the li et al.'s model, our model improves the li et al.'s model by 2.8% in type prediction and 1.5% in value prediction. When compared to vanilla Transformer-xl, we obtain 1.1% improvement in type prediction and 1.8% improvement in value prediction.

From Table 4, we observe that the type prediction accuracies produced by all models are higher than the value prediction accuracies. We attribute this to two aspects: 1) The number of types is far less than the number of values. 2) There is no unknown problem in type.

5.7 Discussion

Why memory mechanism works? The memory mechanism works in two ways. One is the impact of long-dependent learning in our base model, and the other is the impact on the range of copy. The main difference between Transformer-XL and vanilla transformer is the addition of a segment-level recurrence module, which resolves the context fragmentation in vanilla Transformer. In our model, we incorporate the recurrence module into our memory mechanism. This recurrence module allows our model to see more information when dealing with long dependencies. Thus, we observe a performance gain when applying larger memory in our model. Our copy mechanism also depends on our memory mechanism. The hidden states provided by the memory mechanism determines the maximum possible copy length of our copy mechanism. The larger the memory size, the longer the model can copy, and the better the final model performance.

Why copy mechanism works? In the absence of a copy mechanism, all OOV words are regarded as unknown words. When calculating the accuracy, the result of the prediction as unknown needs to be regarded as the wrong prediction result. With a copy mechanism, our model can copy a word from the existing input as a prediction result. The word may be an unknown word or an ordinary word. If it is an ordinary word, it has little effect on the final prediction result since a model without a copy mechanism can also predict it. But if it is an unknown word, the model can additionally increase the accuracy on the existing results. Thus, we observe a performance gain in our model with copy mechanism. In order to verify the effectiveness of our copy mechanism, we replace the parameters related to the copy mechanism that have been trained in our model with randomly initialized parameters, and then performed another experiment. The experimental results are shown in Table 5. The CCMC with randomly initialized parameters and the CCMC without copy mechanism achieve close results, which are not as good as our well-trained CCMC. Thus, we demonstrate that our copy mechanism indeed work in our model.

Table 5: Comparisons between random with trained

	VALUE
CCMC(random copy)	71.44%
CCMC(without copy)	71.45%
CCMC	71.66%

Learning process analysis. To find out why our proposed model get better performance on both type prediction and value prediction, we analyze the learning process of our CCMC and li et al's model, which is the RNN-base state-of-the-art baseline model[18]. As you can see from Figure 6, our CCMC's perplexity is lower than theirs both on train and test throughout the entire learning process. Moreover, they ended with overfitting, and our situation is much better. The difference between training perplexity and testing perplexity of our CCMC has always been relatively small. The reason lies in two aspects: 1) Adopting the Transformer-XL architecture as our base model to model the long-range dependency in the programs helps us obtain a powerful baseline compared to the RNN-based model. 2) Intergrading the pointer network to predict the unknown value helps our model copy the OOV token from the previous context and thus improves the model's performance.

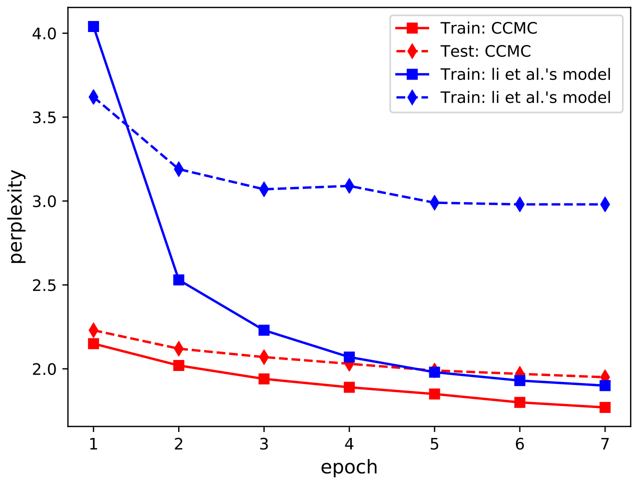
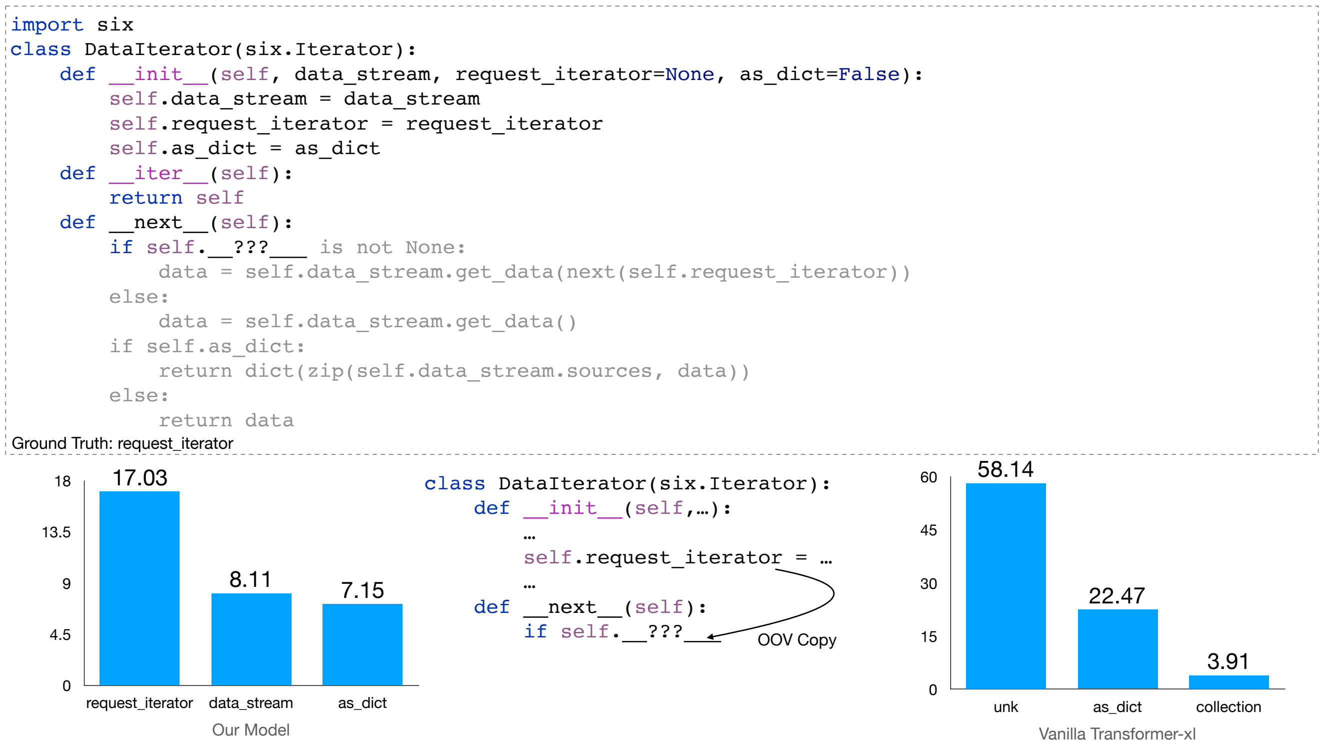


Figure 6: The perplexity on training and test for our CCMC and li et al.'s model

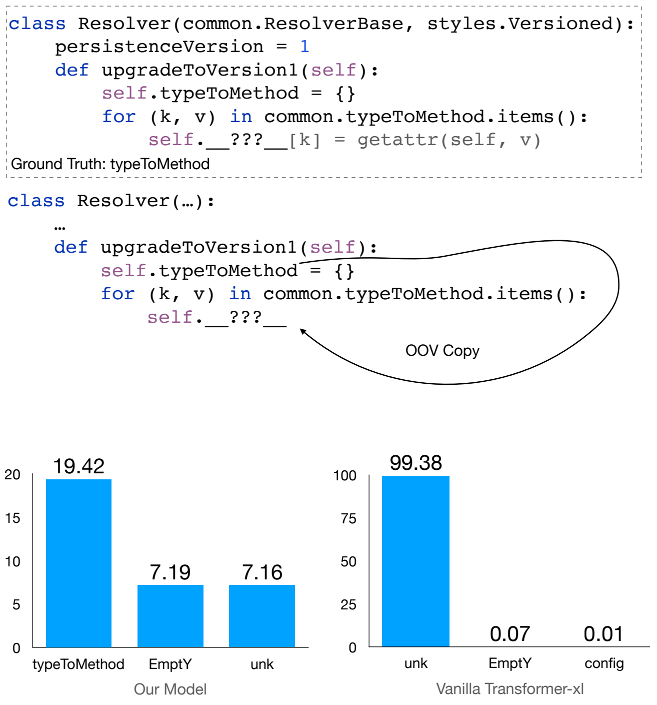
Training cost analysis. To evaluate the cost of our improvement, we analyze the training cost of our CCMC and li et al's model, which is the RNN-base state-of-the-art baseline model[18]. The training time and the number of parameters are shown in Table 6. Compared to li et al.'s model, our CCMC spends 125% of the time and uses 134% of the parameter budget to obtain 2.8% improvement



```
import six
class DataIterator(six.Iterator):
    def __init__(self, data_stream, request_iterator=None, as_dict=False):
        self.data_stream = data_stream
        self.request_iterator = request_iterator
        self.as_dict = as_dict
    def __iter__(self):
        return self
    def __next__(self):
        if self.__???___ is not None:
            data = self.data_stream.get_data(next(self.request_iterator))
        else:
            data = self.data_stream.get_data()
        if self.as_dict:
            return dict(zip(self.data_stream.sources, data))
        else:
            return data
Ground Truth: request_iterator
```

```
class DataIterator(six.Iterator):
    def __init__(self,…):
        …
        self.request_iterator = …
        …
    def __next__(self):
        if self.__???___
```

Figure 7: Example 1. Code completion with OOV value when cross function.



```
class Resolver(common.ResolverBase, styles.Versioned):
    persistenceVersion = 1
    def upgradeToVersion1(self):
        self.typeToMethod = {}
        for (k, v) in common.typeToMethod.items():
            self.__???__[k] = getattr(self, v)
Ground Truth: typeToMethod
```

```
class Resolver(…):
    …
    def upgradeToVersion1(self):
        self.typeToMethod = {}
        for (k, v) in common.typeToMethod.items():
            self.__???__
```

Figure 8: Example 2. Code completion with the OOV value in one function.

in type prediction and 1.5% improvement in value prediction. But considering the improvements, the cost is acceptable.

Table 6: Training cost analysis

	Training Time	Number of Parameters
li et al.'s model.	16h	73M
CCMC	20h	98M

5.8 Case Study

We present code completion examples in Python to demonstrate the effectiveness of the copy mechanism in our model. In our examples, the target prediction `request_iterator` and `typeToMethod` is the OOV value. We show the top 3 predictions of our model and vanilla Transformer-xl. In the first example, see Figure 7, a variable named `request_iterator` is defined in the function `__init__` and used in the function `__next__`. Our model successfully predicts the OOV word, while vanilla Transformer-xl predicts that the result is unknown. In the second example, see Figure 8, a variable named `typeToMethod` is declared and used in the loop. Although vanilla Transformer-xl learns from the context that the target has 99% probability of being unk, it cannot predict the real value correctly. Through our copy mechanism, our model learns to copy the variable from the previously declared location and predict it successfully.

5.9 Threats to Validity

Threats to external validity relate to the quality of the dataset we used and the generalizability of our results. The python dataset PY150 we used is a benchmarked dataset, which contains 150,000 program files in AST formats, with 100,000 used for training and the remaining 50,000 used for testing. All of the programs in PY150 are collected from GitHub. The reasons for using the PY150 dataset are as follows. The python programming language is widely used in software development. A lot of work [18, 21] related to code completion has been launched on this dataset. However, further studies are need to validate and generalize our findings to other programming language.

Threats to internal validity include the influence of the memory size used in our model. The performance of our model would be affected by the different memory size. When we choose the memory size, we double it. Although the memory size increased from 800 to 1600, the performance of the model did not increase. However, our model still achieve a considerable performance compared with Li et al.[18]. Another threat to internal validity relates to the importance of the loss of TYPE and the loss of VALUE. Currently we think they are equally important to our model. When calculating the final loss, they are not given different weights. However, further studies are need to validate the different impact of the loss of TYPE and the loss of VALUE for code completion models.

Threats to construct validity relate to the suitability of our metric. The accuracy we used is the proportion of correctly predicted node. A lot of work [18, 21] related to code completion has used this metric to report their model's performance.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose **C**ode **C**ompletion with **M**emory mechanism and **C**opy mechanism (CCMC). To capture the long-range dependencies in the program source code, we employ Transformer-XL as our base model. To deal with the OOV token in code completion, we integrate the pointer network into our base model and design a memory mechanism and a copy mechanism to combine these two network. Through our copy mechanism, our model can either generate a within-vocabulary token or copy an OOV token from inputs. We also design CopyMask to improve the training efficiency. Experiments on a benchmarked dataset show the superiority of our CCMC model on the code completion task. In the future, we plan to incorporate more compile information[2, 5] such as control flow and data flow to improve our model.

ACKNOWLEDGMENTS

This work has been supported by the National Key R & D Program of China (No. 2018YFB1003800), National Natural Science Foundation of China (No. 61772560), Natural Science Foundation of Hunan Province (No. 2019JJ40388), and the Fundamental Research Funds for the Central Universities of Central South University (No. 2021zzts0744).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[3] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).
[4] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.
[5] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *International Conference on Machine Learning*. PMLR, 1475–1485.
[6] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2018).
[8] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao Wuen Hon. 2019. Unified Language Model Pre-training for Natural Language Understanding and Generation. (2019).
[9] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393* (2016).
[10] Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the unknown words. *arXiv preprint arXiv:1603.08148* (2016).
[11] Sangmok Han, David R Wallace, and Robert C Miller. 2009. Code completion from abbreviated input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 332–343.
[12] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.
[13] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.
[14] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[15] Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. 2018. Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context. *Association of Computational Linguistics (ACL)* (2018).
[16] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code Prediction by Feeding Trees to Transformers. arXiv:2003.13848 [cs.SE]
[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[18] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).
[19] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).
[20] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[21] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016), 731–747.
[22] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.
[23] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).
[24] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.
[25] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.
[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.
[27] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 334–345.