Multi-task Learning based Pre-trained Language Model for Code Completion

Fang Liu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
liufang816@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Yunfei Zhao
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
zhaoyunfei@pku.edu.cn

Zhi Jin*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

Code completion is one of the most useful features in the Integrated Development Environments (IDEs), which can accelerate software development by suggesting the next probable token based on the contextual code in real-time. Recent studies have shown that statistical language modeling techniques can improve the performance of code completion tools through learning from large-scale software repositories. However, these models suffer from two major drawbacks: a) Existing research uses static embeddings, which map a word to the same vector regardless of its context. The differences in the meaning of a token in varying contexts are lost when each token is associated with a single representation; b) Existing language model based code completion models perform poor on completing identifiers, and the type information of the identifiers is ignored in most of these models. To address these challenges, in this paper, we develop a multi-task learning based pre-trained language model for code understanding and code generation with a Transformer-based neural architecture. We pre-train it with hybrid objective functions that incorporate both code understanding and code generation tasks. Then we fine-tune the pre-trained model on code completion. During the completion, our model does not directly predict the next token. Instead, we adopt multi-task learning to predict the token and its type jointly and utilize the predicted type to assist the token prediction. Experiments results on two real-world datasets demonstrate the effectiveness of our model when compared with state-of-the-art methods.

*Corresponding authors.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416591

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

code completion, multi-task learning, pre-trained language model, transformer networks

ACM Reference Format:
Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416591

1 INTRODUCTION

As the complexity and scale of the software development continue to grow, large corpora of open source software projects present an opportunity for modeling source code on machine learning [1]. Most of these approaches are based on the observation of source code's naturalness [19], that is, source code is written by humans and for humans to read, it displays some of the statistical properties as natural language. Thus, statistical language models have been used for source code modeling [19, 42, 46], benefiting many software engineering tasks, including code summarization [23, 49], code clone detection [51, 52] program repair [15, 47], especially, in code completion [17, 19, 27, 46].

Code completion is an essential feature of Integrated Development Environments (IDEs). It speeds up the process of software development by suggesting the next probable token based on existing code. In recent years, as the success of deep learning, Recurrent Neural Network (RNN)-based language models have been applied to source code modeling [3, 27]. In these models, a piece of source code is represented as a source code token sequence or an Abstract Syntactic Tree (AST) node sequence. Given a partial code sequence, the model computes the probability of the next token or AST node and recommends the one with the highest probability. Furthermore, these language models can also learn useful word embeddings, which can be used for other downstream tasks in the same way as word2vec-style embeddings [37]. However, source code has some special properties, which have not been exploited in

```
public long getMaximumTime(IoEventType type) {
    if (!timerManager.containsKey(type))
        throw new IllegalArgumentException("Please add
            this event first.");
    return timerManager.get(type).getMaximum();
}
```

Code 1: A Java method example.

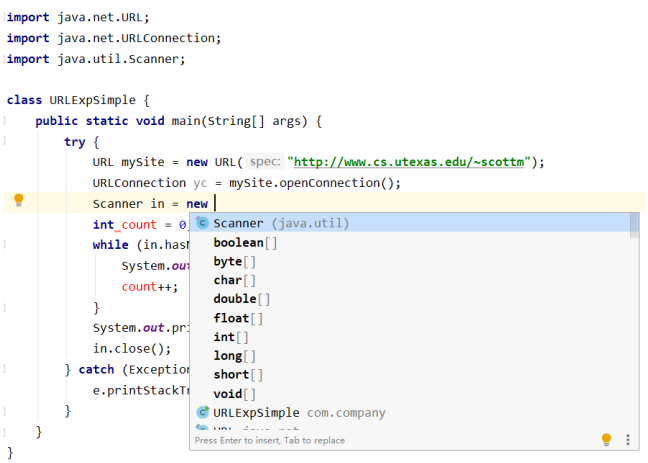

Figure 1: Java IDE completion example.

existing statistical language models. We discuss two critical issues in detail below.

The contextual information is not well considered in the existing code completion models. Writing clean and readable code that conforms to the specification has been paid more attention in software development, which helps the developers reuse and maintain the code. When programming, developers tend to use meaningful and conventional identifier names and natural language documentation [36]. As a result, information contained in the source code can be exploited by machine learning algorithms. Most of these models are based on learned representations called embeddings, which transform words into a continuous vector space [37]. However, existing research [3, 25, 27] uses static embeddings, which map a word to the same vector regardless of its context. For example, in Java method overloading, the same function name can have different meanings based on the number and type of the parameters. However, the static embedding will map it to the same vector. The differences in the meaning of a token in varying contexts are lost when each token is associated with a single representation. The surrounding tokens of the program entities usually contain certain information that reflects the roles of the entities. For instance, for a method name, the surrounding tokens might include the variables/fields/methods that are used/accessed/invoked to implement the method. Taking the Java method in Code 1 as an example, the function name *getMaximumTime* can be inferred from the variables' names and method calls in the body, e.g., *getMaximum*, *timerManager*. These tokens provide information about possible values the function could take, and so should affect its representation.

Identifier completions are challenging, and existing statistical Language Model (LM) based code completion models perform poorly on completing identifiers. These approaches consider every token in the source code file as targets for completion. More than two-thirds of the completions do not refer to identifiers. Instead, the majority concern punctuation-like tokens (e.g., operators, braces), which are much easier to complete than identifiers, but these completions are not that beneficial to developers [25]. Besides, the type information of the identifiers is ignored in most of the models. Modern IDEs for most languages heavily rely on types to make helpful suggestions for completing partial code. For example, when accessing the field of an object in a Java IDE, code completion suggests suitable field names based on the object's type [35]. Taking the code completion example of a Java IDE (ItelliJ IDEA) in Figure 1 as an example, the IDE suggests "Scanner" as the next token based on its type (i.e., java.util), and not just predict the frequent token in the corpus. For those dynamic languages, such as Python and JavaScript, IDEs often fail to make accurate suggestions because the types of code elements are unknown, which further demonstrates the importance of the type information. However, most of the existing LM-based source code modeling techniques and code completion studies do not take the type information into consideration.

In response to the observations and concerns raised above, we have developed a Code Understanding and Generation pre-trained Language Model (CugLM) for source code modeling. Recent work on pre-trained language models has found that the contextual embeddings produced by these models can lead to better performance for many natural language processing (NLP) tasks [10, 22, 39, 40]. In these models, the representation for each word is learned using the language models, where the vector of the word is computed based on the context it is used. Thus, the vector of the same word under different contexts can be different. In particular, BERT [10] proposes a bidirectional Transformer Encoder with two new pre-training objective: "masked language model" and "next sentence prediction", where "masked language model" randomly masks some of the tokens from the input, and the objective is to predict the masked word based only on its context, and "next sentence prediction" predicts whether two sentences follow each other in a natural discourse. By using these two objectives, BERT can produce powerful bidirectional contextual representations and advances the state-of-the-art for many NLP tasks. Inspired by the success of pre-trained language models in NLP, we propose a multi-task learning based pre-trained language model to produce general and contextual representations for programs that can broadly support code understanding and generation tasks, and then apply it to code completion. During the pre-training period, we adopt the multi-task learning framework to learn the following three training objectives jointly:

1) Masked bidirectional Language Modeling: The identifiers are more informative for understanding the program and correctly suggesting the identifiers is challenging in existing code completion research [25]. Thus, producing contextual and general representations for tokens, especially for identifiers, would be helpful for source code modeling and code completion. For these reasons, we mask the identifiers from the programs, and the objective is to predict the masked tokens based on their bidirectional context.

2) Next Code segment Predicting: We argue that understanding relationships between code segments can help in source code modeling. In order to achieve this, we pre-train a binarized next code segment prediction task, that is, predicting whether two segments of code tokens follow each other in a piece of code snippet.

3) Unidirectional Language Modeling: a left-to-right language modeling task, where the representation of each token encodes only the leftward context tokens and itself. This training objective is added because for the generation tasks (e.g., code completion), only leftward contextual tokens are allowed.

After the model has been pre-trained, we fine-tune it (directly apply the pre-trained model and adapt the model on downstream tasks by fine-tuning the pre-trained parameters) on the code completion task. During the code completion, our model does not directly predict next token, instead, we adopt a multi-task learning framework to predict the token and its type. We first predict the type of the token, and then use predicted type to assist the token prediction.

We create two massive corpora of Java and TypeScript programs collected from GitHub to pre-train and fine-tune the model. We compare our model with two state-of-the-art code completion approaches: Byte Pair Encoding based Neural Language Model (BPE NLM) [25] and Pointer Mixture Network [27]. For completing all types of tokens, our model achieves the accuracy of 80% and 81% on Java and TypeScript datasets, respectively, which improves Pointer Mixture Network by 17% and 24%, and improves BPE NLM by 19% and 24%, in terms of relative improvements. For identifier completion, our model achieves the accuracy of 48% and 39%, respectively, which improves Pointer Mixture Network by 29% and 34%, and improves BPE NLM by 11% and 9%, in terms of relative improvements.

The main contributions of this paper are summarized as follows:

- We present the first attempt at pre-training a language model with a transformer-based architecture for code completion.
- We take advantage of the type information to help our model make better suggestions on identifiers.
- We compare our model with state-of-the-art code completion models and evaluate the performance of these models on two real-world datasets. Experimental results demonstrate that our model achieves the best performance compared with the baseline models.

2 BACKGROUND

2.1 Statistical Language Model

Statistical language models capture the statistical patterns in languages by assigning occurrence probabilities to a sequence of words in a particular sequence, which will score an utterance high, if it sounds "natural" to a native speaker, and score low the unnatural (or wrong) sentences. Programming languages are kind of languages that contain predictable statistical properties [19], which can be modeled by statistical language models. Given a token sequence S = $s_1, s_2, ..., s_t$, the probability of the sequence is computed as:

$$p(S) = p(s_1)p(s_2|s_1)p(s_3|s_1s_2), ..., p(s_t|s_1s_2, ..., s_{t-1}) \quad (1)$$

The probabilities are hard to estimate when the number of the context tokens $s_1, s_2, ..., s_{t-1}$ is tremendous. The N-gram model based on the Markov assumption is proposed to address this challenge, where the probability of a token is dependent only on the $n-1$ most recent tokens. N-gram based models have been generally applied to code completion [17, 19, 46]. These models have been proved to capture the repetitive regularities in the source code effectively. In recent years, deep recurrent neural networks, including Long Short-Term Memory (LSTM) [20] and Gate Recurrent Unit (GRU) [6], have shown great performance on modeling programming languages [3, 27, 28]. By using recurrent connections and gate mechanisms, information can cycle inside these networks for a long time, which loosens the fixed context size and can capture longer dependencies than the N-gram model.

However, the introduction of the gating mechanism in LSTMs and GRUs might not be sufficient to address the gradient vanishing and explosion issue fully. To ease this issue, attention mechanisms [2, 48], which add direct connections between long-distance word pairs, are proposed. For example, the Transformer [48] is an architecture based solely on attention mechanism. It uses a multi-headed self-attention mechanism to replace the recurrent layers to reduce sequential computation and capture longer-range dependency. Later, Transformer-XL [8] is proposed by introducing the notion of recurrence into the deep self-attention network. Thus it enables the Transformer networks to capture the very long-term dependency during language modeling.

2.2 Multi-task Learning

Multi-task learning is an approach for knowledge transfer across related tasks. It improves generalization by leveraging the domain-specific information contained in the training signals of related tasks [4]. Through sharing hidden layers among tasks, the model can capture the common features among all the tasks. Furthermore, by preferring the representation that all tasks prefer, the risk of over-fitting is reduced, and the model can be more general to new tasks in the future. Multi-task learning has been successfully used in many fields including natural language processing [10, 14, 31], speech recognition [9] and computer vision [32, 34].

2.3 Pre-trained Language Models

Language model pre-training has shown to be effective for NLP, and has achieved the state-of-the-art results across many NLP tasks [7, 10, 22, 39, 40]. The advantages of the pre-trained model can be summarized as follows: (1 By pre-training on the huge corpus, the model can learn universal representations and help with the target tasks; 2) The pre-trained model can provide a better model initialization, which leads to a better generalization performance on the downstream tasks. 3) Pre-training can be regarded as a kind of regularization to avoid over-fitting on small data. To apply the pre-trained language representations to downstream tasks, the feature-based approaches use the pre-trained representations as additional features [39], and the fine-tuning approaches directly adapt the model on the downstream tasks by simply fine-tuning the pre-trained parameters [10, 40]. Generative Pre-trained Transformer (GPT) [40] and Bidirectional Encoder Representations from Transformers (BERT) [10] are the widely used fine-tuning approach, where BERT has significantly improved the performance of a wide range of natural language understanding tasks. However, the bidirectionality nature of BERT makes it difficult to be applied to natural language generation tasks. To overcome this limitation, UNIfied pre-trained Language Model (UNILM) [11] that can be applied to both natural language understanding (NLU) and natural language generation (NLG) tasks was proposed. Inspired by these models, we build a pre-trained language model for code understanding and generation, and then fine-tune it on code completion.

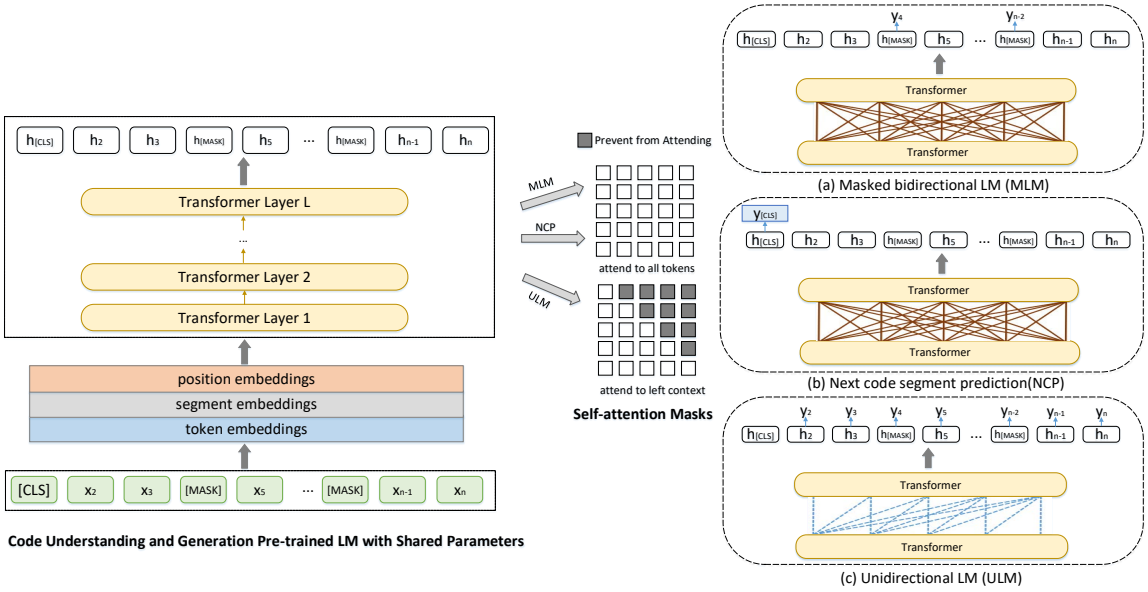

Figure 2: Overview of CugLM pre-training. The model parameters are shared across the pre-training objectives (i.e., MLM, NCP, and ULM). We use different self-attention masks to control the access to context for each token.

3 CugLM

We describe the details about our proposed **C**ode **u**nderstanding and **g**eneration pre-trained **L**anguage **M**odel (**CugLM**) in this section.

3.1 Model Architecture

Given an input program token sequences $x = x_1, x_2, ..., x_n$, CugLM obtains a contextualized vector representation for each token. The model architecture is shown in Figure 2. We adopt an L-layer Transformer as the language model to encode the input vectors $x = x_1, x_2, ..., x_n$ into contextual representations at different levels $H^l = [h_1^l, h_2^l, ..., h_n^l]$, where $H^l = \text{Transformer}_l(H^{l-1}), l \in [1, L]$. In Figure 2 and later sections, we omit the superscript L for the hidden vectors of the final layer h_i^L to make the illustration less cluttered. For each transformer layer (block), multi-attention heads are used to aggregate the output of the previous layer, and the output of a self-attention head A_l is computed as:

$$
\begin{aligned}
&Q = H^{L-1}W_l^Q,\ K = H^{L-1}W_l^K,\ V = H^{L-1}W_l^V \\
&M_{ij} = \begin{cases} 0, \text{ allow to attend} \\ -\infty, \text{prevent from attending} \end{cases} \\
&A_l = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + M)V
\end{aligned}
\tag{2}
$$

where $H^i \in \mathbb{R}^{|x| \times d_h}$ denotes the i-th layer's output. The queries Q, keys K, and values V are computed by linearly projecting the previous layer's output H^{l-1} using parameter matrices W_l^Q, W_l^K, W_l^V. $M \in \mathbb{R}^{|x| \times |x|}$ is the mask matrix that determines whether a pair of tokens can be attended to each other. For different pre-training objectives, we use different mask matrices M to control how many contextual tokens can a token attend to when computing its contextualized representations, as illustrated in Figure 2. For bidirectional LM, the elements of the mask matrix are all 0s, which means that all the tokens have access to each other. For unidirectional LM, the upper triangular part of the mask is set to $-\infty$, indicating that each token can only access the leftward context tokens and itself. The output of CugLM includes (1) contextual vector representation of each input token, and (2) the representation of [CLS], which is short for "CLaSsification" and works as the aggregated sequence representation and can be used for classification tasks.

During the pre-training period, the model's parameters are shared and optimized with several objectives, namely, Masked bidirectional LM, Next Code segment Prediction, and Unidirectional LM. After the model is pre-trained, we can then fine-tune it for downstream tasks. In this paper, we fine-tune CugLM on code completion.

3.2 Input Representation

The input x is a token sequence, which is a pair of segments packed together. As shown in Figure 3, for a given token, its vector representation is computed by summing the corresponding token, segment and position embeddings.

- For token embeddings, the embedding matrix is randomly initialized, and then is adjusted as part of the training process. Two special tokens [CLS], [SEP] are defined, where [CLS], which is short for "CLaSsification", always appears at the beginning of the input. The final hidden state corresponding to [CLS] can be used as the aggregate sequence representation for classification tasks, for example, in next sentence prediction task. [SEP], which is short for "SEPeration", is used to separate the sentence pairs.
- The segment embeddings, i.e., E_A and E_B are also used to differentiate the code segment pairs. For each token of the first code segment, a learned embedding E_A is added, and a learned embedding E_B is added to each token of the second code segment. The embedding matrix for the segment embeddings is also randomly initialized

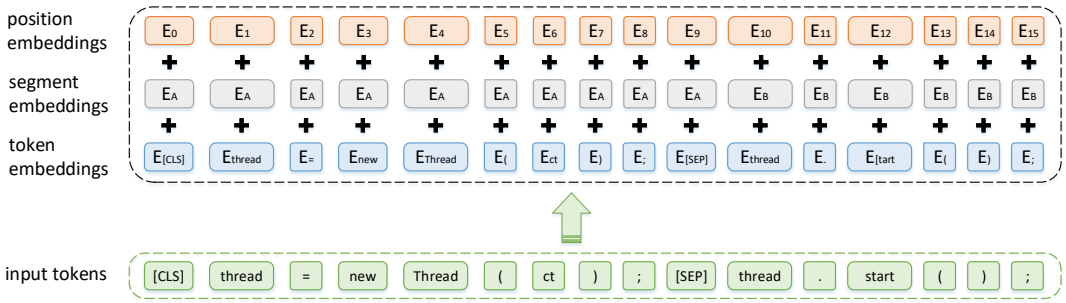

Figure 3: Input representation. The input embeddings is the sum of the token embeddings, the segment embeddings, and the position embeddings.

- To make use of the order of the sequence, we use learned positional embeddings with sequence lengths up to 128 tokens.

3.3 Pre-training Procedure

To pre-train CugLM, we adopt multi-task learning to learn three tasks jointly, as shown in Figure 2, including Masked bidirectional Language Modeling (MLM), Next Code segment Predicting (NCP), and Unidirectional Language Modeling (ULM). For the first two objectives, the Transformer network is under the bidirectional settings, and for the last objective, the Transformer network is unidirectional.

a) **Masked bidirectional Language Modeling**: In order to train deep bidirectional representations for the program, we adopt a similar objective with BERT, that is, masking some percentage of the input tokens and then predicting only those masked tokens. Different from BERT, we only mask the identifiers with type information, where the type information can be extracted by static analysis or be annotated by developers, considering that these identifiers are more informative for understanding the program. Then the objective is to predict the masked identifiers based on their bidirectional contextual tokens, where all tokens can attend to each other in prediction. It encodes contextual information from both directions and can generate better contextual representations of the masked identifiers as well as the other tokens than its unidirectional counterpart. The final hidden vectors corresponding to the mask tokens are fed into the output *softmax* layer to produce the probability distribution of the outputs.

b) **Next Code segment Predicting**: Understanding the relationship between two sentences is quite important for many NLP tasks, for example, Question Answering (QA) and Natural Language Inference (NLI), which can help to understand the input text in more depth. We argue that understanding relationships between code segments also help in source code modeling. In order to achieve this, we pre-train a binarized next code segment prediction task, that is, predicting whether two segments of code tokens follow each other in a piece of code snippet. Specifically, when choosing the code segments A and B for each pre-training example, 50% of the time B is the actual next code segment that follows A, and 50% of the time it is a random code segment from the corpus. For example:

Input = [CLS] public void setTextDirection (int textDirection) { [SEP] this . mTextDirection = textDirection ; }
Label = 1

Input = [CLS] public void setTextDirection (int textDirection) { [SEP] this . request = request ;
Label = 0

The final hidden vector corresponding to [CLS], which works as the aggregated sequence representation, is fed into the output *softmax* layer to produce the probability distribution of classification results.

c) **Unidirectional Language Modeling**: For language generation tasks, for example, code completion, the context of the predicted token should only consist of the token on its left. Thus, we create the left-to-right language modeling task as another pre-training objective, namely predicting the next token x_{t+1} given the preceding context tokens $x_1, x_2, ..., x_t$. The representation of each token encodes only the leftward context tokens and itself. This can be done using a triangular matrix for self-attention mask M, where the upper triangular part of the self-attention mask is set to $-\infty$, and others to 0. At each time step t, the final hidden vector corresponding to x_t is fed into the *softmax* layer to produce the probability distribution of the predicted token y_t.

The pre-training procedure follows the existing language model pre-training approaches. The parameters of CugLM are learned to minimize the sum of the cross-entropy losses of the three pre-training tasks, and are shared among all the tasks. The final loss function is given below:

$$\min_{\theta} \mathcal{L}_{MLM}(\theta) + \mathcal{L}_{NCP}(\theta) + \mathcal{L}_{ULM}(\theta) \tag{3}$$

3.4 Fine-tuning Procedure

When the model is pre-trained, we fine-tune it on code completion task. In code completion, the context of the predicted token should only consist of all the token on its left. Thus, the representation of each token can encode only the leftward context tokens and itself. During the fine-tuning procedure, the following two objectives are optimized:

a) **Unidirectional Masked Language Modeling (UMLM)**: Different from the MLM objective in pre-training, the UMLM objective in fine-tuning is to predict the masked token based only on its leftward context, where all tokens can only attend to the tokens on its left in prediction. The transformer network is set to unidirectional using a triangular matrix for the self-attention mask. All the identifiers that have type information are masked in each sequence.

Besides, our model not directly predicts the masked token. Instead, we adopt the multi-task learning framework to predict the token and its type. We first predict the type of the token, and then the predicted type is used to assist the token prediction, as shown in Figure 4. The reason for formulating the code completion task as a two-step prediction instead of predicting the type and token jointly lies in that, by predicting the type firstly and then use the predicted results as extra input for the token prediction can constraint our model to make more accurate prediction on the type and further enhance the token prediction performance.

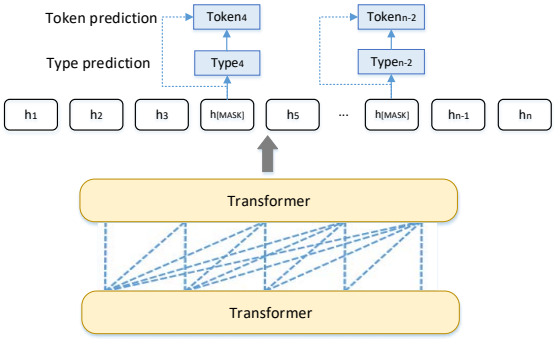

Figure 4: Model architecture for UMLM.

1) Type prediction: The final hidden vector (i.e., the output of the Transformer) corresponding to the mask token $h_{[MASK]}$ is used to compute the output vector for the token's type O_{type}. We use the *softmax* function to produce the probability distribution of the outputs Y_{type}:

$$\begin{aligned} O_{type} &= \tanh(W^o h_{[MASK]}) \\ Y_{type} &= \text{softmax}(W^y O_{type} + b^y) \end{aligned} \tag{4}$$

where $W^o \in \mathbb{R}^{H \times H_{type}}$, $W^y \in \mathbb{R}^{V_{type} \times H_{type}}$, $b^y \in \mathbb{R}^{V_{type}}$ are trainable parameters. V_{type} is the vocabulary size of the token's type, H is the hidden size of the transformer network, H_{type} is the embedding size of type vector.

2) Token prediction: After predicting the token's type, we use the predicted type to assist the token prediction. The vector of the predicted type E_{type} and the hidden vector of the mask token $h_{[MASK]}$ are concatenated to compute the output vector for the token O_{token}. Then the output vector is fed into the output *softmax* layer to compute the output vector for the token Y_{token}:

$$\begin{aligned} O_{token} &= \tanh(W^o(h_{[MASK]}; E_{type})) \\ Y_{token} &= \text{softmax}(W^y O_{token} + b^y) \end{aligned} \tag{5}$$

where E_{type} is the embedding of the predicted type, $W^o \in \mathbb{R}^{H_{token} \times H}$, $W^y \in \mathbb{R}^{V_{token} \times H_{token}}$, $b^y \in \mathbb{R}^{V_{token}}$ are trainable parameters. V_{token} is the vocabulary size of the token, and ";" denotes the concatenation operation.

b) **Unidirectional Language Modeling (ULM)**: This objective is a left-to-right language modeling task that is the same as the pre-training procedure. Given the preceding context tokens $x_1, x_2, ..., x_t$, the model predicts the next token x_{t+1}, where the representation of each token encodes only the leftward context tokens and itself.

Table 1: Statistics of the datasets.

	Java	TypeScript
Projects	9,708	8,446
Files	800,983	227,424
Lines	$5.4 * 10^7$	$8.8 * 10^6$
# of Tokens	$6.9 * 10^6$	$1.1 * 10^6$
# of Types	$6.4 * 10^6$	$1.7 * 10^5$
Masked ID proportion	21.04%	9.74%

During the fine-tuning procedure, the parameters of CugLM are learned to minimize the sum of the cross-entropy losses of the two fine-tuning tasks and are shared among all the tasks. The final loss function is given below:

$$\min_{\theta} \mathcal{L}_{UMLM}(\theta) + \mathcal{L}_{ULM}(\theta) \tag{6}$$

Through learning these two objectives jointly, we hope the model can make better predictions on both the identifiers and the other tokens.

4 EXPERIMENTS AND ANALYSIS

4.1 Data preparation

We pre-train and fine-tune our model across two programming languages: Java and TypeScript. The programs in the corpus are collected from publicly available open-source GitHub repositories by removing duplicate files and project forks. Each program is tokenized into token sequence. The detailed information is shown in Table 1. We use 60% of the projects for pre-training, and 40% of the projects for fine-tuning on code completion task. During the fine-tuning, we split the projects into train/validation/test sets in the proportion 8:1:1. For the other baselines, all the programs used in pre-training and the training programs used in fine-tuning are used as the training set, and the validation and test sets are the same as in our fine-tuning procedure. We also randomly sample 200 program files from both Java and TypeScript test sets as the small test sets for Byte Pair Encoding based Neural Language Model (BPE NLM) [25] evaluation since when performing completion (testing) in their model, they use a variation of the beam search algorithm to combine the sub-units to complete tokens, which is very time-consuming. It takes several minutes to complete a single program file and will take tens of days to perform completion on the large test sets (e.g., the Java test set contains 14,600 files). Thus, we create small test sets.

For Java programs, we extract the identifiers' type information through static analysis. For TypeScript programs, we apply the approach in Hellendoorn et al. [16] to extract type annotations of the identifiers. We filter the programs to make sure at least 10% of type annotations are user-defined types in each TypeScript file. Figure 5 shows the examples for Java and TypeScript code, where the identifiers that have type are marked with underlines, and the green tokens next to the identifiers are the corresponding types. To generate each training input sequence for pre-training, we sample two spans of tokens from the corpus, which we refer to as segments S_1 and S_2. Each segment contains several lines of source code tokens. For the first segment S_1, we sample the first N lines from one program file, where N is randomly sampled from

Table 2: Performance of baseline models and our approach.

Model	Java				TypeScript			
	Large Test		Small Test		Large Test		Small Test	
	All Tokens	Identifiers	All Tokens	Identifiers	All Tokens	Identifiers	All Tokens	Identifiers
Vanilla LSTM	64.28%	33.84%	64.73%	33.11%	64.42%	28.11%	63.31%	23.91%
Pointer Mixture Network	68.30%	38.41%	68.49%	37.54%	68.75%	33.76%	65.75%	29.26%
BPE NLM	-	-	67.17%	43.67%	-	-	65.39%	36.16%
Transformer-XL	72.12%	43.63%	70.96%	40.92%	73.94%	37.46%	68.88%	34.90%
CugLM	84.06%	55.19%	80.07%	48.47%	82.14%	41.85%	81.36%	39.28%

TypeScript

```
type NamespaceName = |'s'|'s'|'s'|'s'|'s'|'s'|'s';
interface Signature {
  name: string;
  email: string;
  when: Date;
}
interface Commit {
  author: Signature;
  committer: Signature;
  sha: string;
  message: string;
}
interface NamespaceInfo {
  count: number;
  namespace: NamespaceName;
  data: { [key]: {
      intro:string;
      name: string;
    };
  };
}
```

Java

```
package com.labo.kaji.swipeawaydialog;
import android.app.Application;
import android.test.ApplicationTestCase;
public class ApplicationTest extends ApplicationTestCase <Application> {
  public ApplicationTest ( ) {
    super(Application.class);
  }
}

ApplicationTestCase: android.test.ApplicationTestCase
Application: android.app.Application
ApplicationTest: com.labo.kaji.swipeawaydialog.ApplicationTest
Application: android.app.Application
```

Figure 5: Code examples for type annotations.

1 to the length of the code lines of the program file. 50% of the time, the second segment S_2 is the rest of the lines from the same program file that follows S_1, and 50% of the time it is a random code segment sampled from other program files of the corpus, which is done for the "next code segment prediction (NCP)" task. They are sampled such that the combined length is ≤ 128 tokens. For the "Masked bidirectional Language Modeling (MLM)" task, we only mask those identifiers that have type information. For example, the underlined tokens in Figure 5.

4.2 Experimental Setup

Parameter configuration. We use Transformer with 6 layers, 516 dimensional hidden states and 6 attention heads. The inner hidden size of the feed-forward layer is 3072. We pre-train our model with batch size of 16 sequences for 600,000 steps. We use Adam with learning rate of 5e-5, β_1 = 0.9, β_2 = 0.999, L2 weight decay of 0.01, learning rate warmup over the first 1,000 steps, and linear decay of the learning rate. We use a dropout probability of 0.1 on all layers. We use a gelu activation [18] following OpenAI GPT. The training loss is the sum of the cross-entropy losses of the pre-training objectives or fine-tuning objectives. Training of CugLM was performed on 3 GeForce GTX 1080 Ti GPUs with 12GB memory. For each dataset, the model is pre-trained for 600,000 steps and takes 4 days to complete, and is fine-tuned for 300,000 steps and takes 2 days to complete.

Metric. We use *accuracy* to evaluate the performance of code completion. Our model provides an ordered list of suggestions for each token in the source code file given the context. We compute the top-1 accuracy, i.e., the fraction of times the correct suggestion appears in the first of the predicted list.

Vocabulary. As shown in Table 1, in the datasets, the number of unique tokens and types is too large to build neural models to learn directly. We choose K (50,000) most frequent tokens in each training set to build the token vocabulary, which is the same as Li et al. [27]'s study. For those tokens outside the vocabulary, we use *UNK* (unknow values) to represent them. The size of type vocabulary is also set to 50,000. In both the training and test process, the predictions of the *UNK* targets are treated as wrong predictions. The token *UNK* rates for Java, and TypeScript test sets are 10%, 5%, and the type *UNK* rates are 9%, 1%, respectively.

4.3 Research Questions and Results

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

RQ1: How does our proposed approach perform in code completion when compared with state-of-the-art models? To answer this research question, we compare our model with the following baseline models:

- vanilla LSTM: a vanilla LSTM neural network-based language model.
- Pointer Mixture Network [27]: an attention and pointer-generator network-based code completion model.
- Byte Pair Encoding based Neural Language Model (BPE NLM) [25]: a large-scale open-vocabulary NLM for code completion, which leverage BPE [13] algorithm to keep vocabulary size low and successfully predict OoV (Out-of-Vocabulary) tokens.
- Transformer-XL [8]: a self-attentional neural network-based language model for code completion.

Table 3: Effects of each pre-training task, fine-tuning task, and the type prediction in our proposed model.

Model	Java				TypeScript			
	Large Test		Small Test		Large Test		Small Test	
	All Tokens	Identifiers	All Tokens	Identifiers	All Tokens	Identifiers	All Tokens	Identifiers
Full Model	84.06%	55.19%	80.07%	48.47%	82.14%	41.85%	81.36%	39.28%
Pre-training tasks								
- ULM	78.64%	50.10%	77.78%	44.18%	77.83%	38.44%	76.77%	37.38%
- MLM	77.42%	49.86%	76.41%	43.82%	78.93%	36.89%	78.28%	35.15%
- NCP	81.24%	52.56%	79.79%	46.54%	78.52%	40.71%	79.02%	38.49%
Fine-tuning tasks								
- UMLM	80.93%	45.70%	77.21%	41.66%	78.86%	33.26%	77.58%	31.81%
- ULM	-	49.50%	-	43.31%	-	38.25%	-	35.33%
- Type Prediction	80.14%	52.05%	77.28%	46.83%	80.99%	40.73%	79.85%	38.31%

1) Comparison with LSTM based closed vocabulary models (the first two baselines): To compare with Pointer Mixture Network, we downloaded their publicly available source code[1]. In their model, the programs in the datasets are parsed into ASTs, and they build the model to perform code completion on AST node sequences. Although the ASTs can provide more information, representing the programs as AST node sequences is not the natural order of typing, and the precision does not directly reflect the productivity gain of the code completion tool. More importantly, in practice, the code is incomplete, so the software project might not be compilable (code is not parsable into ASTs, or parsed ASTs miss a lot of information). Thus, representing programs as token sequences and performing code completion on the token-level might be more practical. In this paper, we focus on token-level code completion. In our corpus, the programs are tokenized into token sequences. To compare with them, we train their model within our tokenized programs using the command line arguments given in the artifact's README file [2]. Their base model is a single layer LSTM network with an unrolling length of 50 and hidden unit size of 1500. The initial learning rate is 0.001 and is decayed by multiplying 0.6 after every epoch. The gradients' norm is clipped to 5. The size of the attention window is 50. Since the Pointer Mixture Network is based on LSTM language model, we also list the results of the vanilla LSTM, where the parameter configuration of the vanilla LSTM network is set the same as the Pointer Mixture Network.

As shown from the results, our model outperforms the two LSTM-based models on both Java and TypeScript datasets by a large margin, especially in identifier completion. On the Java large test set, our model achieves the accuracy of 84.06% and 55.19% on token's completion and identifier's completion, respectively, which outperforms Pointer Mixture Network by 23.07% and 43.69%, in terms of relative improvement. On the TypeScript large test set, our model achieves the accuracy of 82.14% and 41.85% on token's completion and identifier's completion, respectively, which outperforms Pointer Mixture Network by 19.47% and 23.96%. The results on small test sets are similar to the large test set. We can find that the improvements on the TypeScript dataset are smaller than Java, especially in identifier completion. The reason lies in that, the (masked) identifier proportion in TypeScript (9.74%) is smaller than Java (21.04%) because the type information in TypeScript is annotated by developers, and only a part of the identifiers are annotated. In the MLM pre-training task, these identifiers are masked and are predicted based on their contextual tokens aiming at generating better contextual and informative representations for these identifiers as well as other tokens. During fine-tuning, the type information of these identifiers is used to assist the identifiers' prediction. Due to the lower masked proportion, the pre-training and fine-tuning procedure can offer less information than Java, thus resulting in smaller improvements.

2) Comparison with open vocabulary model (BPE NLM): To compare with BPE NLM, we downloaded their publicly available source code[3] and train their model on our datasets. They use a single layer GRU NLM with an unrolling length of 200 built upon sub-word units learned from BPE. The embedding size and the hidden unit size are both set to 512 in their model. To keep the number of parameters comparable with our model and other baselines, we increase the hidden unit size and the embedding size of their model to 1500. There are three scenarios: static, dynamic, and maintenance, where the dynamic and maintenance settings update model's parameters during testing. Since our model and other baselines do not update parameters during the test process, we present the results of the static scenario to make the comparison fair, and realize that evaluating dynamically may improve accuracy. As shown from the results, BPE NLM performs best on completing identifiers among all the baseline models on both datasets, which proves the power of the open vocabulary LM for predicting the identifiers. Even though, our model still outperforms the BPE NLM on completing identifiers. When evaluating on completing all kinds of tokens, the performance of BPE NLM is not as well as the identifier completion. Our model outperforms BPE NLM on completing all kinds of tokens by a large margin.

3) Comparison with transformer network based model (Transformer-XL): To find out if CugLM's promising results derive more from using a Transformer-based model for code completion, or from the

[1] https://github.com/jack57lee/neuralCodeCompletion

[2] Since the Pointer Mixture Network also makes use of the additional information derived from ASTs, the results of using token sequence as input might understate the accuracy of the plain Pointer Mixture Network.

[3] https://github.com/mast-group/OpenVocabCodeNLM

multi-task learning based pre-training and fine-tuning, we compare our results to a Transformer-based model trained from scratch, i.e., without the benefit of a pre-trained embedding. Transformer-XL is a Transformer network based language model, which introduces the notion of recurrence to model the long-term dependency of the input sequence. We use a 6-layer Transformer-XL network with 5 parallel heads. The dimension of each head is set to 64. We set the segment length to be 128, and the length of the cached segments to 256. The dimensionality of the model (hidden unit) and the embedding size is set to 800. The dimension of the feed-forward layer is set to 1024. As seen from Table 2, transformer-XL model outperforms the other baseline models that are based on the recurrent neural networks on both datasets, which demonstrates that the Transformer-based network is more powerful than recurrent neural networks on this task. The performance of our model is substantially higher than the Transformer-XL model trained from scratch. We therefore conclude that pre-training and fine-tuning are crucial to CugLM's success.

RQ2: What are the contributions of the pre-training tasks? We perform an ablation study to examine the effects of the three pre-trained tasks: ULM, MLM, and NCP. We conduct experiments on pre-training the model without each task, and the fine-tuning procedure remains unchanged. The results are shown in Table 3. The first row shows the results of our full model. The second to the fourth rows present the results of removing ULM, MLM, and NCP from the full model during pre-training, respectively.

- **ULM** Removing the ULM task during pre-training. The loss function of the pre-training procedure consists of $\mathcal{L}_{MLM}$ and $\mathcal{L}_{NCP}$, and both these tasks are based on the bidirectional transformer. As seen from the results, removing this task hurts the model's performance. During fine-tuning, the objectives are based on the unidirectional transformer. Thus, adding the ULM task during pre-training makes the learned text representations more general because they are optimized for both bidirectional and unidirectional language modeling objectives jointly, mitigating over-fitting to bidirectional language modeling task. Removing the ULM task would make the parameters hard to optimized when fine-tuned on the unidirectional objectives. Thus, the accuracy drops.
- **NCP** Removing the Next Code segment Prediction task during the pre-training. The loss function consists of $\mathcal{L}_{ULM}$ and $\mathcal{L}_{MLM}$. The NCP tasks are added to help the model understand the relationships between the code segments. The model removing NCP performs worse than the full model, but performs better than removing ULM, which demonstrates that the NCP task is necessary to improve the performance but contributes less than the ULM task.
- **MLM** Pre-training the model without the Masked bidirectional Language Modeling objective, and the loss function consists of $\mathcal{L}_{ULM}$ and $\mathcal{L}_{NCP}$. As shown from the results, removing the MLM hurts the performance more than the other two tasks, especially on identifier completion. MLM task can help the model generate better contextual representations of the tokens, especially the identifiers, thus can improve the model's performance significantly.

The above results demonstrate that all of the pre-training tasks are necessary to improve the performance, and MLM contributes most to the improvements.

RQ3: What are the contributions of the fine-tuning tasks? To figure out the effectiveness of the fine-tuning procedure, we also conduct experiments by removing each of the fine-tuning task. The results are shown in fifth and sixth rows of Table 3.

- **UMLM** Removing the Unidirectional Masked Language Modeling task during fine-tuning procedure. Only the left-to-right language modeling task is performed and the loss function becomes $\mathcal{L}_{ULM}$. As seen from the results, removing this task hurts the model's performance on both two datasets, especially for the identifier prediction. UMLM task can help the model generate better contextual representations for the tokens. Besides, it can also utilize the type information of the identifiers during the fine-tuning. Thus, this fine-tuning task is necessary for improving the performance of the code completion.
- **ULM** Removing the Unidirectional Language Modeling task during fine-tuning procedure. Under this setting, the model can only produce the results of the masked identifier prediction. The loss function becomes $\mathcal{L}_{UMLM}$. As seen from the results, when removing ULM task, the performance of the identifier prediction drops a lot, which demonstrates that the language modeling task can offer much help for the identifier prediction. Through optimizing the model on this task jointly, the model can capture the semantic of the input code segment better, which serves as the basis of the improvement on identifier prediction.

RQ4: Could the predicted type help the model on token prediction? When fine-tuning our model on code completion task, we utilize multi-task learning to predict the token and its type jointly. We first predict the type and then use the type to assist the token's prediction. To confirm whether our model can correctly predict the identifier's type, we present the accuracy of the type prediction. Our model achieves the accuracy of 68.89% and 79.31% on Java and TypeScript large test sets, respectively. The results demonstrate that our model can correctly predict the identifiers' type in most cases. To find out whether the type prediction really helps, we conduct experiment by removing the type prediction. The results are shown in the last row of Table 3. As shown from the results, when removing the type prediction, the model performs worse than the full model on completing both identifiers and all tokens, which demonstrates that the predicted type information can help the model achieve better performance on code completion.

5 DISCUSSION

5.1 The type of completions

Except for identifiers, we also give a detailed breakdown of the accuracies for completing different types of tokens on both our model and BPE NLM [25], and also present these tokens' proportion. The results are shown in Table 4. Punctuations make up the majority of the completions, and the accuracies of both our model and BPE NLM on predicting punctuations are high, where BPE NLM performs better than CugLM. The punctuation tokens are much easier to complete than identifiers, but these completions are not that useful for developers [25]. For keyword completions, our model outperforms BPE LM by a large margin. The keywords are predefined, reserved words used in programming that have special meanings to the compiler, which contain the syntactic information or the attribute information of the objects. The great performance of CugLM on completing keywords further demonstrates that through multi-task learning based pre-training and

Table 4: Performance of completing different types of tokens.

	Java			TypeScript		
Type	Proportion	CugLM	BPE NLM	Proportion	CugLM	BPE NLM
Identifiers	28.99%	**48.47%**	42.27%	16.62%	**39.28%**	36.16%
Keyword	7.69%	**86.78%**	72.57%	6.49%	**79.47%**	56.67%
Punctuation	31.98%	87.38%	**90.30%**	45.42%	82.64%	**82.93%**
Numerals	0.62%	**72.62%**	58.83%	1.22%	**89.42%**	82.44%
Operator	3.80%	**85.65%**	76.92%	4.03%	**75.98%**	65.84%

fine-tuning, the representations generated by our model can capture syntactic and semantic information better. For numeral and operator completions, which are more related to the semantic of the programs, our model also outperforms BPE NLM substantially.

5.2 Model complexity comparison

To analyze the complexity of our model and the baseline models, we present the number of trainable parameters for all the models, shown in Table 5. The number of trainable parameters of our model is less than all the baselines. Although we adopt multi-task learning for both pre-training and fine-tuning, the number of trainable parameters does not increase much as all of the tasks share one multi-layer transformer network. To improve training efficiency and avoid over-fitting, we do not use large parameter settings. Even though, our model still outperforms the other baselines by a large margin thanks to the pre-training and fine-tuning.

Table 5: Parameters of the baseline models and our model.

Model	# of Parameters
Vanilla LSTM	168M
Pointer Mixture Network	177M
BPE NLM	145M
Transformer-XL	173M
CugLM	104M

5.3 Effect of applying BPE algorithm

To further improve the performance of our model, we also conduct experiments on applying Byte Pair Encoding (BPE) algorithm to build up the vocabulary of sub-words as in [25], where the rare tokens will be segmented into more common sub-word units, and no word is OoV. However, the performance on Java corpus is comparable with the origin model, and the accuracy decreases slightly on TypeScript corpus. We analyze the possible reasons are as follows. During pre-training, we mask the identifiers with type information. When we apply BPE algorithm, most of these masked identifiers will be split into sub-word units. Thus, all of these units will be masked, which leads to the high mask proportion and increased the difficulty of learning the semantics of embeddings. Besides, during fine-tuning, our model utilizes the predicted type information to assist the token's prediction. After splitting the tokens into sub-word units, all of the units from one token correspond to the same type, resulting in the semantic inconsistencies between the type information and the sub-word units. For example, the same unit from different tokens might correspond to different types. Thus, applying BPE does not improve the performance of our model.

5.4 Threats to Validity

Threats to external validity relate to the quality of the datasets we used and the generalizability of our results. We create two massive datasets (Java and TypeScript) to pre-train and fine-tune our model. All of the programs in the datasets are collected from GitHub repositories. The reasons for using these two languages are as follows. These two languages are commonly used for software development, and we can get the identifiers' type through static analysis or through the developers' annotations. However, further studies are needed to validate and generalize our findings to other programming languages.

Threats to internal validity include the influence of the hyper-parameters used in our model. The performance of our model would be affected by different hyper-parameter settings, which are tuned empirically in our experiments. Thus, there is little threat to the hyper-parameter choosing, and there might be room for further improvement. However, current settings have achieved a considerable performance increase. Another threat to internal validity relates to the implementation of the baseline methods. For Li et al. [27]'s model, we apply their model to the token-level code completion, which is originally used for AST-level code completion. In their model, the additional information derived from ASTs is utilized to improve the performance. The results of using token sequence as input might understate the accuracy of the plain Pointer Mixture Network. However, in practice, the code is incomplete, so the code is not parsable into ASTs, or parsed ASTs miss a lot of information. Thus, representing programs as token sequences and performing code completion on the token-level is more practical. Under this setting, we have tried our best to make fair comparison with Li et al. [27] by only changing the format of the input, and keeping the model unchanged. For BPE NLM [25], we compare our model with the static setting of their model considering the fairness of the comparison. We realize that evaluating dynamically may improve accuracy. The dynamic and maintenance scenarios are not implemented and compared in this work, which will be considered as our future work.

Threats to construct validity relate to the suitability of our evaluation measure. We use *accuracy* as the metric which evaluates the proportion of correctly predicted next token. It is a classical evaluation measure for code completion and is used in almost all the previous code completion work [17, 19, 27, 41, 46].

6 RELATED WORK

Statistical Code Completion Code completion is a hot research topic in the field of software engineering. Early work in code completion mainly bases on heuristic rules and static type information

to make suggestions [21]. Since Hindle et al. [19] found that source code contained predictable statistical properties, statistical language models began to be used for modeling source code [17, 27, 30, 38, 50], where N-gram is the most widely used model. [46] observed that source code has a unique property of localness, which could not be captured by the traditional N-gram model. They improved N-gram by adding a cache mechanism to exploit localness and achieved better performance than other N-gram based models. Hellendoorn and Devanbu [17] introduced an improved N-gram model that considered the unlimited vocabulary, nested scope, locality, and dynamism in source code.

In recent years, deep recurrent neural network-based language models have been applied to learning source code and have made great progress [3, 5, 27–29]. Liu et al. [28] proposed a code completion model based on a vanilla LSTM network. Li et al. [27] proposed a pointer mixture network to address the OoV issue. Liu et al. [29] propose a multi-task learning and transformer based language model for AST-level code completion. They built model to predict the AST node's type and value jointly and also utilized the hierarchical structural information in the program's representation, which achieves state-of-the-art results on AST-level code completion. Kim et al. [26] presented a transformer model for code prediction and incorporated syntactic structure into the transformer to further improve the model's performance. Svyatkovskiy et al. [44] proposed a code completion system based on LSTM for recommending Python method calls. Their system is deployed as part of the Intellicode extension in Visual Studio Code IDE. Karampatsis et al. [25] proposed a large-scale open-vocabulary neural language model for source code, which leverages the BPE algorithm, beam search algorithm, and cache mechanism to both keep vocabulary size low and successfully predict OoV tokens. The experimental results demonstrate that their open vocabulary model outperforms both N-gram models and closed vocabulary neural language models, and achieve state-of-the-art performance on token-level code completion. Most recently, Svyatkovskoy et al. [45] implemented and evaluated a number of neural code completion models, which offer varying trade-offs in terms of memory, speed and accuracy. They provided a well-engineered approach to deep-learning based code completion, which is important to the software engineering community.

Pre-trained Language Models Language model pre-training has shown to be effective for NLP, and has achieved the state-of-the-art results across many NLP tasks [7, 10, 22, 39, 40]. Pre-trained language models can learn token contextualized representations by predicting tokens based on their context by training on large amounts of data, and then can be adapted to downstream tasks. Bidirectional Encoder Representations from Transformers (BERT) [10] is the widely used approach in NLP, which learns to predict the masked words of a randomly masked word sequence given surrounding contexts. BERT has significantly improved the performance of a wide range of natural language understanding tasks. Kanade et al. [24] extended this idea to programming language understanding tasks. They derived contextual embedding of source code by training a BERT model on source code. They evaluate their model on a benchmark of five classification tasks in programs. Results show that their model outperforms the baseline LSTM models supported by Word2Vec embeddings, and Transformers trained from scratch. The bidirectionality nature of BERT makes it difficult to be applied to natural language generation tasks. To overcome this limitation, Dong et al. [11] proposed a unified pre-trained language model (UNILM) that can be applied to both natural language understanding and natural language generation tasks. UNILM can be configured using different self-attention masks to aggregate context for different types of language models, and thus can be used for both language understanding and generation tasks.

In the above work, the models are learned from the input of a single modal, for example, only from natural languages or source code. In recent years, multi-modal pre-trained models that can learn implicit alignment between inputs of different modalities are proposed. These models are learned from bi-modal data, such as pairs of language-image [33], language-video [43], or language-code [12]. Feng et al. [12] proposed CodeBERT, a bimodal pre-trained model for natural language and programming language, aiming at capturing the semantic connection between natural language (NL) and programming language (PL). They trained CodeBERT with masked language modeling task and replaced token detection task, and evaluated it on two downstream NL-PL tasks, including natural language code search and code documentation generation.

Inspired by the above models, we propose a code understanding and generation pre-trained language model with a transformer-based architecture and tailored it for code completion, which is the first attempt at pre-training a language model for code completion.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a multi-task learning based code understanding and generation pre-trained language model for source code modeling with a Transformer-based neural architecture. We pre-train our model on two massive datasets and with three objective functions and then fine-tune it on code completion. Experimental results demonstrate that the proposed model achieves better results than previous state-of-the-art models on completing tokens, especially on completing identifiers. To the best of our knowledge, we are the first to apply the pre-trained language model to code completion. We believe this work represents a significant advance in source code modeling, which will be beneficial as a building block for many other applications in this area.

In the future, we plan to apply our model to other programming languages and fine-tune our model to adapt to other tasks.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program under Grant No. 2018YFB1003904, and the National Natural Science Foundation of China under Grant Nos. 61832009, 61620106007, and 61751210.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. (2015).
[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] Rich Caruana. 1997. Multitask Learning. *Machine Learning* 28, 1 (1997), 41–75.

[5] Hao Chen, Triet Huynh Minh Le, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges. *ACM Computing Surveys (CSUR)* (2020).

[6] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. (2014), 103–111.

[7] Andrew M Dai and Quoc V Le. 2015. Semi-supervised sequence learning. In *Advances in neural information processing systems*. 3079–3087.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. 2978–2988.

[9] Li Deng, Geoffrey E. Hinton, and Brian Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: an overview. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 8599–8603.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[11] 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. In *Advances in Neural Information Processing Systems*. 13042–13054.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[13] Philip Gage. 1994. A new algorithm for data compression. *C Users Journal* 12, 2 (1994), 23–38.

[14] Han Guo, Ramakanth Pasunuru, and Mohit Bansal. 2018. Soft Layer-Specific Multi-Task Summarization with Entailment and Question Generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 687–697.

[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[16] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162.

[17] 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.

[18] Dan Hendrycks and Kevin Gimpel. 2016. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. (2016).

[19] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 837–847.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[21] Daqing Hou and David M Pletcher. 2010. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. 26–30.

[22] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *ACL (1)*. Association for Computational Linguistics, 328–339.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. 200–210.

[24] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained Contextual Embedding of Source Code. *arXiv preprint arXiv:2001.00059* (2019).

[25] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. ICSE.

[26] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code Prediction by Feeding Trees to Transformers. *arXiv preprint arXiv:2003.13848* (2020).

[27] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 4159–4165.

[28] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural Code Completion. (2016).

[29] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Ming Li, Zhiyi Fu, and Zhi Jin. 2019. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. *arXiv preprint arXiv:1909.06983* (2019).

[30] Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented LSTM. *Journal of Systems and Software* 164 (2020), 110547.

[31] Xiaodong Liu, Jianfeng Gao, Xiaodong He, Li Deng, Kevin Duh, and Ye-Yi Wang. 2015. Representation Learning Using Multi-Task Deep Neural Networks for Semantic Classification and Information Retrieval. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*. The Association for Computational Linguistics, 912–921.

[32] Mingsheng Long and Jianmin Wang. 2015. Learning Multiple Tasks with Deep Relationship Networks. *CoRR* abs/1506.02117 (2015). arXiv:1506.02117 http://arxiv.org/abs/1506.02117

[33] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In *Advances in Neural Information Processing Systems*. 13–23.

[34] Yongxi Lu, Abhishek Kumar, Shuangfei Zhai, Yu Cheng, Tara Javidi, and Rogério Schmidt Feris. 2017. Fully-Adaptive Feature Sharing in Multi-Task Networks with Applications in Person Attribute Classification. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 1131–1140.

[35] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[36] Robert C. Martin. 2009. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall.

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

[38] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 532–542.

[39] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *NAACL-HLT*. Association for Computational Linguistics, 2227–2237.

[40] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf* (2018).

[41] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM, 731–747.

[42] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 419–428.

[43] Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. 2019. Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE International Conference on Computer Vision*. 7464–7473.

[44] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[45] Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. 2020. Fast and Memory-Efficient Neural Code Completion. *arXiv preprint arXiv:2004.13651* (2020).

[46] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 269–280.

[47] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720* (2019).

[48] 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.

[49] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[50] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing*

Systems. 6563–6573.

[51] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[52] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.