electronics

Article

Improving Text-to-Code Generation with Features of Code Graph on GPT-2

Incheon Paik [1,*] and Jun-Wei Wang [2]

1 School of Computer Science and Engineering, The University of Aizu, Fukushima 965-8580, Japan
2 Department of Computer Science and Information Engineering, ChaoYang University of Technology, Taichung 413310, Taiwan; z6228574@gmail.com
* Correspondence: paikic@u-aizu.ac.jp; Tel.: +81-242-37-2796

Abstract: Code generation, as a very hot application area of deep learning models for text, consists of two different fields: code-to-code and text-to-code. A recent approach, GraphCodeBERT uses code graph, which is called data flow, and showed good performance improvement. The base model architecture of it is bidirectional encoder representations from transformers (BERT), which uses the encoder part of a transformer. On the other hand, generative pre-trained transformer (GPT)—another multiple transformer architecture—uses the decoder part and shows great performance in the multilayer perceptron model. In this study, we investigate the improvement of code graphs with several variances on GPT-2 to refer to the abstract semantic tree used to collect the features of variables in the code. Here, we mainly focus on GPT-2 with additional features of code graphs that allow the model to learn the effect of the data stream. The experimental phase is divided into two parts: fine-tuning of the existing GPT-2 model, and pre-training from scratch using code data. When we pre-train a new model from scratch, the model produces an outperformed result compared with using the code graph with enough data.

Keywords: code generation; data flow; BERT; AST; GPT-2

Citation: Paik, I.; Wang, J.-W. Improving Text-to-Code Generation with Features of Code Graph on GPT-2. *Electronics* **2021**, *10*, 2706. https://doi.org/10.3390/electronics10212706

Academic Editor: George A. Tsihrintzis

Received: 17 September 2021
Accepted: 2 November 2021
Published: 5 November 2021

1. Introduction

Deep learning has advanced the performance of models for code generation significantly. The full automatic code generation contributes to solving problems in the shortage of resources for application development in various business domains. There have been many approaches to improve text-to-code generation performance such as the sequence-to-tree model with attention [1], dual task of code summarization [2], and pre-training code representations with data flow (GraphCodeBERT) [3]. The current state-of-the-art approach, GraphCodeBERT, trained CodeXGLUE data on bidirectional encoder representations from transformers (BERT) together with a code graph of which semantic data flow was extracted from the data set. The GraphCodeBERT uses the inherent structure of the code instead of using the AST code structure in the pre-training phase. The semantic code structure is used to let the code understand the relationship between variables and variable types. Because the code is executed from top to bottom, it is dependent on the context. However, the convenience of BERT regeneration is not particularly good. If BERT is used for code generation, it needs to be matched with the seq2seq architecture. In the experimental phase, we discuss BERT code generation in detail.

BERT uses the encoder part of the transformer. On the other hand, generative pre-trained transformer (GPT)—another multiple transformer architecture—uses the decoder part and has shown great achievement as a language model with generative pre-training of a language model on a diverse corpus of unlabeled text, followed by discriminative fine-tuning.

In this study, we investigate the improvement of efficiency by code graphs with several variances on GPT-2. We refer to the Abstract Syntax Tree (AST) used by GraphCodeBERT to collect the features (variables and variable type status) in the code. The purpose of this study is to use the collected variables and variable types to make different arrangements while GrapthCodeBERT uses data flow with a unique variable concept. Based on the different arrangements to use the basic ontological concept of data flow, we expect that GPT-2 produces different effects and we will observe them from experiments.

GPT [4–6] uses the decoder part of the transformer, and longer and larger data sets can be used. The demonstration of the GPT uses a large number of corpora and shows a brilliant result on general function Q&A, writing articles and translation, and code generation. The GPT-2 has good context capturing ability and semantic information possessed by data flow. We believe that the effect of these two methods on code generation can improve the performance. We carry out this experiment in two phases.

- First, we use the model that did not use a code graph in the pre-training phase. Next, the code graph is used to view the effect on the model during fine-tuning.
- A model is pre-trained from scratch and uses the code graph for training based on stage one. Using this model, we investigate the performance of the pre-training and fine-tuned models.

In summary, the contributions of this paper are: (1) We propose a pre-trained and fine-tuned model of text and code with several features of variables, and a pre-trained model from scratch based on GPT-2. (2) The detailed variable features with basic ontological concept of data flow are trained and their effect is observed. (3) Effects of the several variable conditions are evaluated.

2. Related Work

Code generation is a relatively new topic in the deep learning area that has gained some attention over the last few years. Long short-term memory (LSTM), transformer, and BERT have achieved good results. The recently published GPT-3 has made a breakthrough in code generation. However, most of them are code predictions. Code prediction assists programmers in programming and gives tips on writing programs at appropriate times. Our study allows the model to generate a whole string of codes at one time, such as function or class.

2.1. Code Generation by LSTM

In this section, we present seven studies that used LSTM to complete the code generation and showed good performance. The first study [7] allowed the LSTM model to learn the structure of the code. The code splits small nodes according to the brackets. If a forward parenthesis is encountered, the node is split into another small node. The system stops after encountering the last code or anti-brackets. The second study [2] used the relationship between code-to-text and text-to-code. In that study, both comment code and code comment generation were trained. At the same time, the system calculated the attention value of these two models and the probability value of the last hidden layer, and used the language model to obtain the probability value of the original comment and code. The six values obtained by the two models were used as the final double constraint and, finally, brought back to the original model to obtain the result. Another study [8] used the LSTM model to generate the shell code for the text description. Continuously, this method works for another language, [9,10] used Python code-based code generation and LSTM's encoder and decoder. Finally, the studies with code structure [11,12] mainly used the AST method to analyze the structure of the code.

2.2. Code Generation by Transformer

In transformers, Sun et al. built an encoder–decoder with six layers each of the encoder and decoder and a total of 12 layers of encoder–decoder. The authors used the transformer's [13] attention mechanism to alleviate the long-dependency problem and introduced a novel AST encoder to incorporate grammatical rules and AST structure into the model. The TreeGen is evaluated according to the Python-based HearthStone and two semantic analysis benchmarks (ATIS and GEO).

However, these works leverage the model to learn models on term sequence arrangement without pre-training.

2.3. Code Generation by Pre-Trained Models

For the BERT, Guo et al. [3] also used AST to convert the code into a tree structure. The nodes of this structure are the variable sequence data. Most of them are composed of variables in the program. The authors input the obtained comment, code, and obtained variables into the BERT re-pretrain. According to the re-pretrain described by the authors, the obtained variables can be linked with the original code. The final results improved the effectiveness of the authors' various tasks. Mark Chen et al. [14] evaluated a large set of language models trained on code using GPT-3 architecture. Guo et al. [15] proposed a BERT model to text-to-SQL generation with content enhancement. Norouzi et al. [16] uses BERT to evaluate learning ability to generate code from natural language with less prior knowledge but more monolingual data.

However, our work uses GTP-2 model with diverse variable features with base ontological characteristics for improvement.

2.4. Code Generation Model with Code Structure

Allamanis et al. [17] use graph neural networks to reason over program structures with graphs for program code. Hellendoorn et al. [18] propose a model using gated graph neural network and transformer to combine local or global data to enable more detailed program structure in code. The GraphCodeBERT [3] uses BERT model with text, code, and uniform data flow information.

However, this work uses GPT-2 pre-training model with several variable features with basic ontological information.

3. Data and GraphCodeGPT

For this study, we used the GPT-2 model. A transformer is a model composed of encoding and decoding. However, if the transformer is disassembled into separate encode and decode, it becomes two different models. Encode is a model BERT released by Google, and decode is the GPT-2 model released by OpenAI. Both models have grown substantially in various assessments. There are also large-scale natural language models such as Transformer XL and XL Net. In particular, the performance in the context of the generated text has exceeded original expectations, and adding a large number of data sets to the training of the model is the greatest contributor.

3.1. Data Preparation

3.1.1. Data Set

The CodeSearchNet data set provided by CodeXBLUE is used for training and testing. We explain the data for the CodeSearchNet data set. It is one of CodeXBLUE's data platforms that contain many issues related to code, and uses the most basic code search, code translation, and code generation items. In CodeXBLUE, there are four main items: text-to-text, text-to-code, code-to-text, and code-to-code. The most relevant to this study are text-to-code and code-to-text. As shown in Table 1, it includes multiple programming languages (Python, PHP, GO, Java, JavaScript, and Ruby).

Table 1. CodeSearchNet data set.

Programming Language	Train	Dev	Test
Python	251,820	13,914	14,918
PHP	241,241	12,982	14,014
Go	167,288	7325	8122
Java	164,923	5183	10,955
JavaScript	58,025	3885	3291
Ruby	24,927	1400	1261

Each programming language contains the following information for users:

- repo: the owner/repo;
- path: the full path to the original file;
- func_name: the function or method name;
- original_string: the raw string before tokenization or parsing;
- language: the programming language;
- code/function: the part of the original_string that is code;
- code_tokens/function_tokens: tokenized version of code;
- docstring: the top-level comment or docstring, if it exists in the original string;
- docstring_tokens: tokenized version of docstring.

3.1.2. Data Set Description

We divide the data preprocessing into two types. One is to restrict the length of comments and codes, and the other is to remove the structure of the code.

- Restricting the length of comments and code: It allows the model to add the data flow information and to generate complete code at one time. As shown in Table 2, we observe the two data sets. Excessively long comments and codes may result in poor performance and failure to generate complete codes. Therefore, the comment is limited to a character length of 40 and a code character length of 400, as shown in Table 3.

Table 2. Average length of count unrestricted comment and code character.

	Unrestricted	
Examples	**Comment**	**Code**
181,061	176.6	607.3

Table 3. Number of restricted comments and code character length. The comment character average length is limited to 40. The code character length is limited to 400.

	Restricted	
Examples	**Comment**	**Code**
91,735	19.8	126.8

- Removing the structure of the code: The code with structure is fine-tuned in the model, although the inherent structure can be generated when the code is regenerated. As shown in Table 4, we found that the generated code is affected by the linefeed symbols (carriage return, \r), (line feed, \n), and (tabulate, \t) present in the original code in Windows. These symbols allow the code to have a structure that is easy to read, but these symbols are not very important in the model. In the training set, we only use the Java programming language. Therefore, removing this symbol will not affect the execution of the code.

Table 4. Original and visual format of the code.

Original	protected void unsetVirtualHost(VirtualHost vhost) { if (TraceComponent.isAnyTracingEnabled() && tc.isEventEnabled()) { Tr.event(tc, "Unset vhost: ", vhost); } secureVirtualHost = null; }
Visualization	protected void unsetVirtualHost(VirtualHost vhost) {\r\n\tif (TraceComponent.isAnyTracingEnabled() && tc.isEventEnabled()) {\r\n\t\tTr.event(tc, \"Unset vhost: \", vhost);\r\n\t}\r\n\tsecureVirtualHost = null;\r\n}

3.1.3. Data Reconstruction

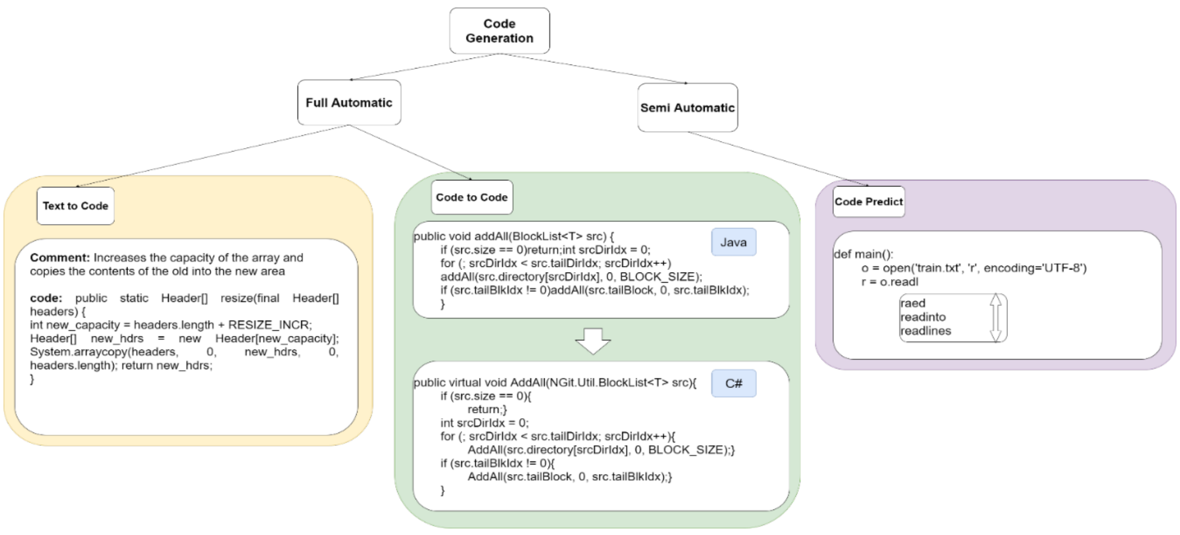
The data are rearranged according to limitations on the length of comments and codes presented in Section 3.1.1. Limiting the length reduces the training data of the model. However, GPT-2 must use a large amount of training data for fine-tuning for a good effect. The training set, development set, and test set are reallocated. Table 5 shows the distribution of reorganized CodeSearchNet data that allows the model to obtain more training data. Our development set takes 10% of the original data set, the test set only uses 1000 data, and the rest is used for model training.

Table 5. Number of data by CodeSearchNet data reconstruction.

CodeSearchNet Total	Train	Dev	Test
91,735	81,561	9174	1000

3.2. Code Generation

As shown in Figure 1, code generation is a field composed of two different topics: semiautomatic and full automatic generation. Most of the code generation is semiautomatic. 'Semiautomatic' means that when the code is entered, the system will recommend the user code [1,19]. The full automatic code generation can be divided into two areas: code-to-code and text-to-code. Code-to-code is similar to language translation of two different programming languages [20], while text-to-code [21,22] can take many forms. The model can be applied to a question to predict the answer, and the other is to give a general and clear purpose and generate a general structure.



Comment: Increases the capacity of the array and copies the contents of the old into the new area

```
code: public static Header[] resize(final Header[] headers) {
int new_capacity = headers.length + RESIZE_INCR;
Header[] new_hdrs = new Header[new_capacity];
System.arraycopy(headers, 0, new_hdrs, 0, headers.length); return new_hdrs;
}
```

```
public void addAll(BlockList<T> src) {
    if (src.size == 0)return;int srcDirIdx = 0;
    for (; srcDirIdx < src.tailDirIdx; srcDirIdx++)
    addAll(src.directory[srcDirIdx], 0, BLOCK_SIZE);
    if (src.tailBlkIdx != 0)addAll(src.tailBlock, 0, src.tailBlkIdx);
    }
```

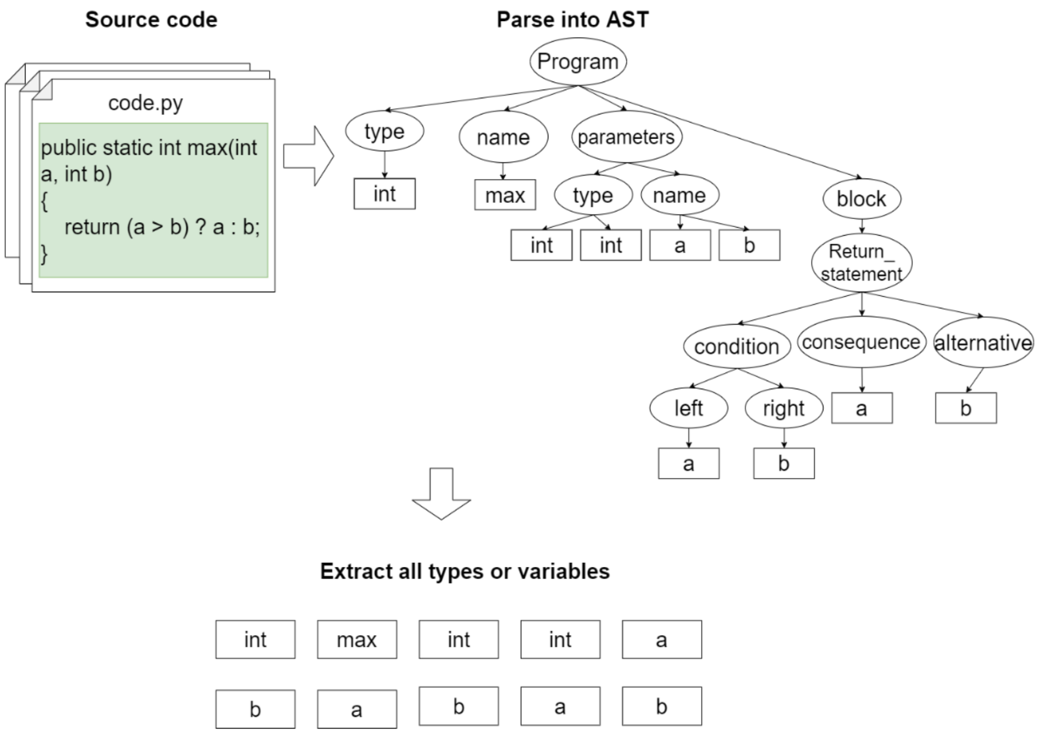
```
public virtual void AddAll(NGit.Util.BlockList<T> src){
    if (src.size == 0){
        return;}
    int srcDirIdx = 0;
    for (; srcDirIdx < src.tailDirIdx; srcDirIdx++){
        AddAll(src.directory[srcDirIdx], 0, BLOCK_SIZE);}
    if (src.tailBlkIdx != 0){
        AddAll(src.tailBlock, 0, src.tailBlkIdx);}
    }
```

```
def main():
    o = open('train.txt', 'r', encoding='UTF-8')
    r = o.readl
            raed
            readinto
            readlines
```

Figure 1. Field of code generation.

3.3. Data Flow

The data flow can represent the dependency graph between variables, variable types, and objects as basic variable ontology. The nodes represent variables, variable types, and objects, and the edges represent where each value comes from. We use an AST to extract the required data flow. AST is an abstract representation of the original syntax structure. It presents the grammatical structure of the programming language in the structure of a tree. As shown in Figure 2, first, we bring a large amount of source code to the AST. Then, we filter the code. Everyone has a different style of writing programs. Therefore, the variable name will be different. Here, we prefer to use the algorithm code for training. The code of the algorithm has a certain regularity that can be traced. For GPT-2, a context-dependent language model, the grammatical structure of the algorithm can effectively help with the training. We prefer generating functions rather than generating complete code. Because of GPT-2 input restrictions, we choose a short code with a clear meaning. All the nodes obtained by the syntax tree generated by AST are features that can be used for the training, and arranging all data streams meaningfully can make GPT-2 focus more on data flows.



```
public static int max(int
a, int b)
{
    return (a > b) ? a : b;
}
```

Figure 2. Data flow flowchart.

3.4. GraphCodeGPT

As shown in Figure 3, we introduce the model architecture of GraphCodeGPT that uses GPT as the base model. Here, we use some code as the natural language on the pre-trained language model with data flow that is a graph similar to the AST tree structure to This graph is show the dependency relationship between variable sequences. Given a Java source code $S = \{s_1, s_2, \ldots, s_n\}$ with its textual content $T = \{t_1, t_2, \ldots, t_m\}$, the corresponding data flow $F(C) = (V, E)$ can be obtained, where $V = \{v_1, v_2, ..., v_k\}$ is a set of variables and $E = \{e_1, e_2, \ldots, e_i\}$ is a set of direct edges that describe where the value of each variable appears. We construct the textual comment from source code and the set of variables as the sequence input $I = \{[CLS], T, [SEP], S, [SEP], V\}$, where *[CLS]* is a special token, and we classify the type of variable as *Type(V) = {Variable No Sort; Variable Sort; Variable type; Variable type add object}*.

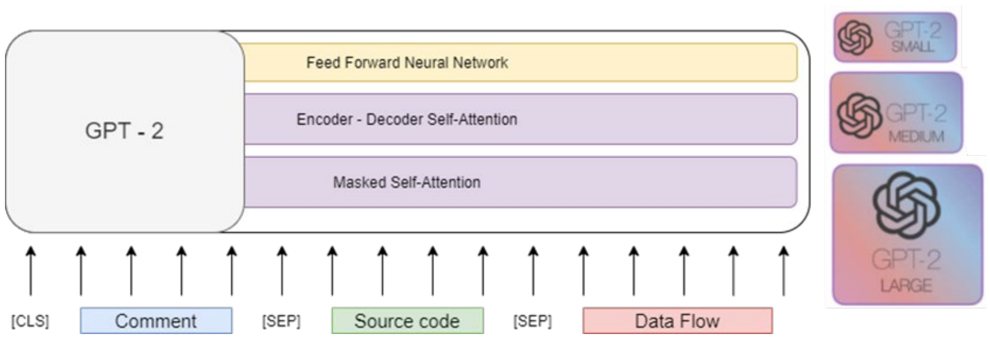


Figure 3. GraphCodeGPT architecture.

GraphCodeGPT takes sequence I as the input and then converts the sequence into input H_0. For each token, its input vector is constructed by summing the corresponding token and position embeddings. We use a special position embedding for all variables to indicate that they are nodes of data flow. The GPT-2 model has N layers of transformers according to different pre-training models. Each layer of transformer [23] contains a self-attention mechanism. The GPT-2 model input context tokens use multi-head self-attention. The output distribution of self-attention generated tokens is as follows, where $U = (u_{-k}, \ldots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

$$h_0 = UW_e + W_p \tag{1}$$

$$h_l = transformer\_block(h_{l-1}) \tag{2}$$

$$P(u) = softmax\left(h_n W_e^T\right) \tag{3}$$

The previous layer's output H_{n-1} is linearly projected to a triplet of queries, keys, and values using model parameters W_i^Q, W_i^K, and W_i^V, respectively. The u represents the number of heads, dk represents the dimension of a head, and W_i^O represents the model parameters. M is a mask matrix, where M_{ij} is 0 if the ith token is allowed to attend the jth token; otherwise, M_{ij} is $-\infty$.

$$Q_i = H_{n-1}W_i^Q,\ K_i = H_{n-1}W_i^K,\ V_i = H_{n-1}W_i^V \tag{4}$$

$$head_i = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M\right)V_i \tag{5}$$

$$\hat{G}^n = [head_i; \ldots; head_n]W_n^O \tag{6}$$

4. Experiment and Evaluation

4.1. Evaluation Metrics

In the evaluation method, we use bilingual evaluation understudy (BLEU) and exact match (EM). BELU is evaluated using Formula (7), where BP represents the penalty factor, to penalize sentences that are too short in length and to prevent the training results from tending to produce sentences that are too short. P_n represents the accuracy of the n-gram. The BLEU of the experimental results is presented as 4-g results. EM means that the model answer matches any standard answer and counts as 1; otherwise, it is zero. Finally, the evaluation system will count the overall accuracy rate.

$$BLEU = BP \times \exp\left(\sum_{n=1}^{N} W_n \log P_n\right) \tag{7}$$

4.2. Fine-Tuning

In the fine-tuning phase, we use the "CodeGPT-Small-Java" pre-trained model provided by CodeXGLEU. As shown in Figure 4, this model only uses annotations and codes for pre-training. In addition, the model uses the same training set we use for pre-training.

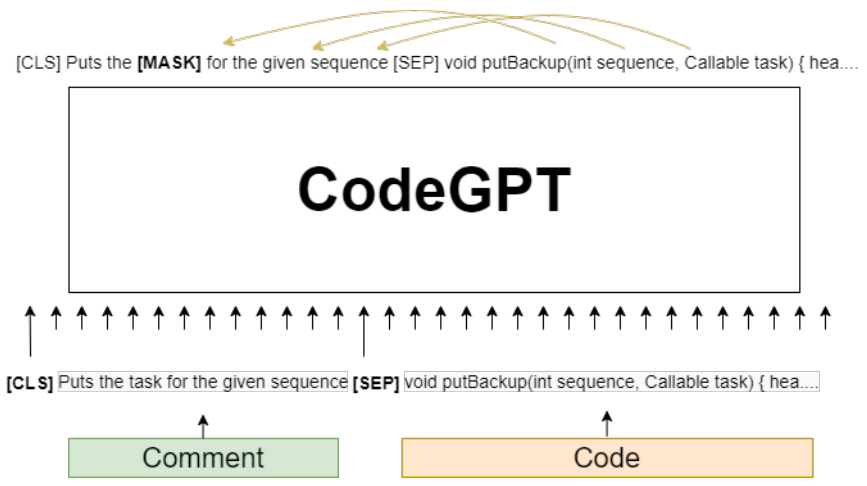

[CLS] Puts the [MASK] for the given sequence [SEP] void putBackup(int sequence, Callable task) { hea....
CodeGPT
[CLS] Puts the task for the given sequence [SEP] void putBackup(int sequence, Callable task) { hea....
Comment
Code

Figure 4. CodeGPT pre-training architecture.

4.2.1. Fine-Tuning without Code Graph

In this section, we evaluate experiments that did not use the code graph. The evaluation is based on BLEU and EM scores. Table 6 shows that the results of 87.11 and 79.2 can be obtained by fine-tuning BLEU and EM without using the code graph. The next stage of the experiment uses this score as the standard score.

Table 6. BLEU and EM scores without code graph.

	BLEU	**EM**
No Code Graph	87.11	79.2

4.2.2. Fine-Tuning with Code Graph

In this section, we evaluate all the code graph methods. 'Variable nosort' is the name of all variables in the data flow. We do not sort these variable names. The reason is that we believe that the variable names that the program executes from top to bottom must be sorted according to the original order. 'Variable sort' is to gather all the same codes together. Thus, when the code is training, the programs of the same code should be linked together again. Variable type and add object use the idea of variable no-sort and add more information. The BLEU and EM scores are also used to evaluate the results. As shown in Table 7, the scores of all BLEU and EM decrease when the code graph is added. However, we do not think that a decrease in scores means bad results.

Table 7. BLEU and EM scores with code graph.

	BLEU	**EM**
Variable Nosort	82.76	76.1
Variable Sort	63.26	51.2
Variabletype	82.48	75.7
Variabletype Add_object	64.24	51.4

According to the data presented in Table 7, we provide an analysis in Table 8. From the results of Examples 1 and 2, the result of the no code graph can show that most of the results are similar to the standard answer. Therefore, similar results lead to higher BLEU. The code in Example 3 indicates that nothing produces the same result as the standard answer. The performance of variable_nosort has some influence on the result because of the influence of the code graph.

Table 8. Comparison of examples of code generation with and without code graph.

Examples	Model	Code
1	GOLD	public boolean searchBase(String base, String filter) throws NamingException { return search(base, filter, scopeBase); }
	No Code Graph	public boolean search(String base, String filter) throws NamingException { return search(base, filter, scopeSub); }
	Variable Nosort	public boolean search(Scope s) {if (defaultScope == null) { defaultScope = s; return true; } return defaultScope.search(s); }
2	GOLD	void format(StringBuilder sb, Temporal localDate) {//Instant localDate = ClockCurrent.GMT.instant(); int len = _timestamp.length; for (int j = 0; j < len; j++) { _timestamp[j].format(sb, localDate); } }
	No Code Graph	public void format(StringBuilder sb, Temporal localDate) {//Instant localDate = ClockCurrent.GMT.instant(); int len = _timestamp.length; for (int j = 0; j < len; j++) { _timestamp[j].format(sb, localDate); } }
	Variable Nosort	public static String format(Integer year, Integer month, Integer day){Object[] _args = {year, month, day};return SHORT__DATE.format(_args);}
3	GOLD	protected void addOption(ApplicationOption<?> option){if(option! = null){this.getCli().addOption(option);this.options.add(option);}}
	No Code Graph	protected void addOption(ApplicationOption<?> option){if(option! = null){this.getCli().addOption(option);this.options.add(option);}}
	Variable Nosort	public void addOption(ApplicationOption<?> option){if(option! = null){this.getCli().addOption(option);this.options.add(option);}}

From the description in Table 8, joining code graph affects all scores. The table shows how it is affected by variable_nosort and variabletype, whose scores are not drastically low. Table 9 shows the "Protected" and "private" results of variable_nosort. This is the correct answer at the beginning of Example 1 because comment does not specify what type to use as the beginning. Consequently, there may be errors, but they will not cause the code to fail to execute. Example 2 produces an error at the beginning of "Long", thus causing the following code to also produce an error because GPT-2 has a very powerful context processing ability. Example 3 is similar to Example 1 and is easily affected by the code graph.

Table 9. Examples of code generation with code graph.

Examples	Model	Code
1	GOLD	protected void reConnectAsync (IOException originalExcep) { setAllowReconnect (false); this.reconnecting = true; this.reconnectException = originalExcep; connectAsynch (getTargetAddress()); }
	Variable Nosort	private void reConnectAsync (IOException originalExcep) { setAllowReconnect (false); this.reconnecting = true; this.reconnectException = originalExcep; connectAsynch (getTargetAddress()); }
2	GOLD	@Override public final DoubleStream limit(long maxSize) { if (maxSize < 0) throw new IllegalArgumentException (Long.toString (maxSize)); return SliceOps.makeDouble (this, (long) 0, maxSize); }
	Variable Nosort	@Override public final LongStream limit (long maxSize) { if (maxSize < 0) throw new IllegalArgumentException(Long.toString (maxSize)); return SliceOps.makeLong (this, 0, maxSize); }
3	GOLD	private static Optional < ICondition > asNotLessThan (JsonObject json) { return Optional.of (json) .filter (j -> j.containsKey (NOT_LESS_THAN_KEY)) . map (j -> j.getJsonObject (NOT_LESS_THAN_KEY)) .map (a -> new AssertNotLess(a.getString (PROPERTY_KEY), a.getInt (MIN_KEY))); }
	Variable Nosort	private static Optional < ICondition > asNotLessThan (JsonObject json) { return Optional.of (json) .filter (j -> j.containsKey (NOT_LESS_THAN_KEY)) .map (j -> j.getJsonObject (NOT_LESS_THAN_KEY)) .map (a -> new AssertNotLess (a.getString (PROPERTY_KEY), a.getInt (MAX_KEY))); }

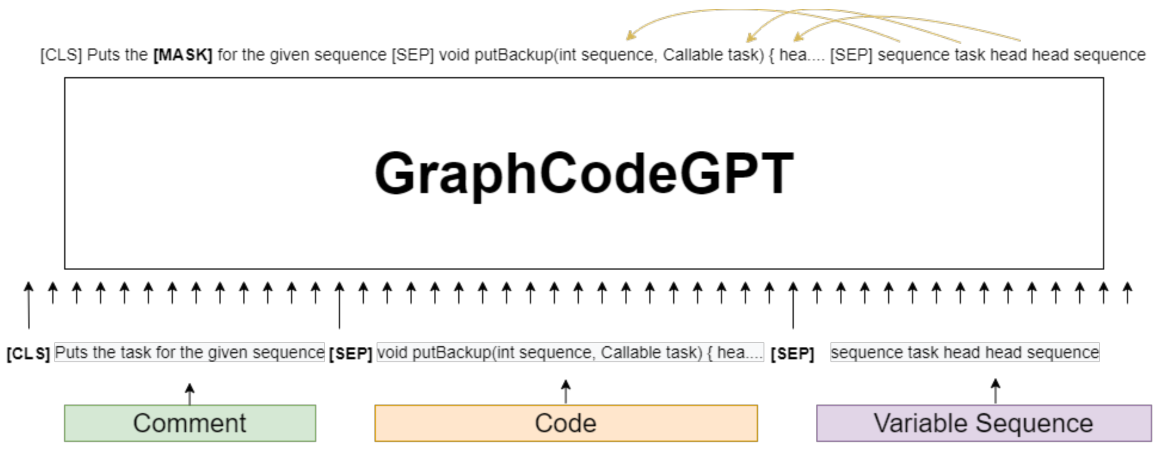
Table 10 presents the special code in the examples generated by GPT-2. The standard answer and the generated answer of Examples 1 and 2 are slightly different. However, variable_nosort uses the code graph method to learn to use another method to generate different codes and the same answer. Consequently, the model can generate a variety of codes according to its own ideas. In addition, if the code graph is not used, it will be close to the standard answer.

Table 10. Special examples of code generation with code graph.

Examples	Model	Code
1	GOLD	public static int distanceSq (int x0, int y0, int x1, int y1) { return ((x1 − x0) * (x1 − x0)) + ((y1 − y0) * (y1 − y0)); }
	Variable Nosort	public static int distanceSq (int x1, int y1, int x2, int y2) { x2 -= x1; y2 -= y1; return x2 * x2 + y2 * y2; }
2	GOLD	private String removeQuotes (String str) { if (str != null && !str.isEmpty()) { str = str.replaceAll("\"", ""); } return str; }
	Variable type	public static String removeQuotes (String s) { if (s.startsWith("\"") && s.endsWith("\"")) return s.substring(1, s.length() - 1); else return s; }

4.3. *Pre-Training Code Graph from Scratch*

As shown in Figure 5, we have trained a GPT model from scratch to allow GPT-2 to show its functions. This pre-training model contains a sequence of variables not shown in Figure 4. The CodeSearchNet data set is also used for pre-training and for reorganizing the data set in Section 3.1.3. The purpose is to use unknown data for development and testing during pre-training. In Section 4.3.1, we discuss the details of the pre-trained model. Section 4.3.2 then fine-tunes the model pre-trained in Section 4.3.1 once.



[CLS] Puts the [MASK] for the given sequence [SEP] void putBackup(int sequence, Callable task) { hea.... [SEP] sequence task head head sequence
GraphCodeGPT
[CLS] Puts the task for the given sequence [SEP] void putBackup(int sequence, Callable task) { hea.... [SEP] sequence task head head sequence
Comment
Code
Variable Sequence

Figure 5. GraphCodeGPT pre-training architecture.

4.3.1. Pre-Training GPT-2 with and without Code Graph

In this section, we use a data set containing comments, codes, and variable sequences to pre-train a new GPT-2 pre-training model. We observe the two cases: The pre-trained model that does not use code graph and the pre-trained model that uses code graph again. These two pre-trained models are divided into three epochs and 30 epoch models for a simple comparison using pre-trained data from scratch. Using the pre-trained model of code graph, we select variable_nosort that performs well in Section 4.2.2.

4.3.2. Fine-Tuning with and without Code Graph

We fine-tune the four pre-trained models trained in the previous section. Table 11 shows the results of 30 epochs' fine-tuning of four pre-trained models. The results show that there is a large difference between the scores of only three epochs of pre-training and 30 epochs of pre-training. Finally, the result of the same 30 epochs shows that the effect of using the code graph is better than that of not using the code graph.

Table 11. Comparison results of the pre-trained model from scratch with and without code graph.

Models	**Epochs**		**BLEU**	**EM**
	Pre-Training	**Fine-Tuning**		
No Code Graph	3	30	10.2	22.44
	30		36.5	52.12
Variable_nosort	3		10.7	22.39
	30		45.6	59.15

The results show a difference between pre-training and fine-tuning when each has 30 epochs. The BLEU and EM of variable_nosort are significantly higher than the no code graph. Because of the analysis based on the results in Table 12, in most cases, no code graph will generate less code. As shown in Example 1, the code of no code graph can completely generate the code of the upper and lower half. However, the intermediate code is missing. As shown in Example 2, the 'void' model from the beginning was affected by unknown data, resulting in an error. The last half of the code also generates less 'return'. The presented reasons lead to the decline in performance in the case of the no code graph.

Table 12. Comparison results of the new pre-trained model with and without code graph.

Examples	**Model**	**Code**
1	Gold	public static FastForward setup (InetAddress addr, int port) throws SocketException { final DatagramSocket socket = new DatagramSocket(); return new FastForward (addr, port, socket); }
	No Code Graph	public static FastForward setup (InetAddress addr, int port) throws SocketException { (final DatagramSocket socket = new DatagramSocket();) return new FastForward (addr, port, null); }
	Variable_nosort	public static FastForward setup (InetAddress addr, int port) throws SocketException { final DatagramSocket socket = new DatagramSocket(); return new FastForward (addr, port, socket); }
2	Gold	public final boolean defineSynthesizedProperty (String propertyName, JSType type, Node propertyNode) { return defineProperty (propertyName, type, false, propertyNode); }
	No Code Graph	public void defineSynthesizedProperty (String propertyName, JSType type, Node propertyNode) { (return) defineProperty (propertyName, type, false, propertyNode); }
	Variable_nosort	public final boolean defineSynthesizedProperty (String propertyName, JSType type, Node propertyNode) { return defineProperty (propertyName, type, false, propertyNode); }

As shown in Table 13, we increase the number of pre-training data in this experiment. The pre-training data use all the data of CodeSearchNet. To test the effect of pre-training, we gradually increase from the original 30 epochs to 300 epochs. The results show that the BLEU and EM scores obtained by only training the model for 30 epochs after increasing the data improved slightly. By contrast, after 100 epochs of training, the model's scores greatly improved. However, the final scores obtained by the model during continuous training for 200 epochs to 300 epochs surpassed the BLEU and EM scores of the no code graph. Thus, this experiment shows that increasing the number of data and the number of pre-training epochs can effectively improve the performance of the model.

Table 13. Result of improvement by pre-training from the scratch.

	Epochs		BLEU	EM
	Pre-Training	Fine-Tuning		
Variable_nosort	30	30	32.5	18
	100		76.17	65
	200		84.81	77.1
	300		88.68	82

5. Conclusions

In this paper, we present GrapchCodeGPT that uses data flow from code. This approach shows an efficient model with GPT-2 similar to BERT. In this work, we mainly used the data type of data flow and added the architecture of the deep learning model GPT-2 to improve code generation. The experimental phase is divided into two phases: fine-tuning and pre-training. In the first stage, we use the existing model and add information about the code graph for fine-tuning. The generated code after fine-tuning was not ideal, so we analyzed the generated code examples to get the idea of the second stage of pre-training. It is not sufficient to add the information of the code graph during the fine-tuning. The code graph pre-training must be added during the pre-training from scratch. The experiment at this stage shows that adding a code graph is very effective. The score of the no code graph in the first experiment was significantly lower than the score of the variable nosort case when they both had 30 epochs. The next experiment increases the number of pre-training data and epochs. The facts have proven that this experiment is correct. When the number of rounds increased to 300 epochs, the results of BLEU and EM surpassed the results of the first phase of the no code graph. The final results show that the score of the model with the code graph is significantly higher than the model without the code graph. The analysis shows the pre-trained GPT-2 model with data flow information increases code generation performance efficiently with enough data.

Author Contributions: J.-W.W. performed data collection, code rewriting, deep learning model construction and training, and results analysis. I.P. suggested the idea of code generation on GPT-2 with the code graph, proposed the method of constructing code generation, and pre-trained the model from scratch. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The information can be obtained through CodeXGLUE github code to text (https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text) (accessed on 18 June 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kim, S.; Zhao, J.; Tian, Y.; Chandra, S. Code Prediction by Feeding Trees to Transformers. *arXiv* **2020**, arXiv:2003.13848.
2. Wei, B.; Li, G.; Xia, X.; Fu, Z.; Jin, Z. Code Generation as a Dual Task of Code Summarization. *arXiv* **2019**, arXiv:1910.05923v1.
3. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv* **2021**, arXiv:2009.08366v3.
4. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. *arXiv* **2020**, arXiv:2005.14165v4.
5. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving Language Understanding by Generative Pre-Training. 2018. Available online: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf (accessed on 23 July 2021).
6. Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language Models are Unsupervised Multitask Learners. 2019. Available online: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (accessed on 23 July 2021).
7. Dong, L.; Lapata, M. Language to Logical Form with Neural Attention. *arXiv* **2016**, arXiv:1601.01280v2.
8. Liguori, P.; Al-Hossami, E.; Cotroneo, D.; Natella, R.; Cukic, B.; Shaikh, S. Shellcode IA32: A Dataset for Automatic Shellcode Generation. *arXiv* **2021**, arXiv:2104.13100v3.
9. Cruz-Benito, J.; Vishwakarma, S.; Martin-Fernandez, F.; Faro, I. Automated Source Code Generation and Auto-completion Using Deep Learning: Comparing and Discussing Current Language-Model-Related Approaches. *arXiv* **2021**, arXiv:2009.07740v4.
10. Yin, P.; Neubig, G. A Syntactic Neural Model for General-Purpose Code Generation. *arXiv* **2017**, arXiv:1704.01696v1.
11. Parvez, M.R.; Chakraborty, S.; Ray, B.; Chang, K.A. Building Language Models for Text with Named Entities. *arXiv* **2018**, arXiv:1805.04836v1.
12. Rabinovich, M.; Stern, M.; Klein, D. Abstract Syntax Networks for Code Generation and Semantic Parsing. *arXiv* **2017**, arXiv:1704.07535v1.
13. Sun, Z.; Zhu, Q.; Xiong, Y.; Sun, Y.; Mou, L.; Zhang, L. TreeGen: A Tree-Based Transformer Architecture for Code Generation. *arXiv* **2019**, arXiv:1911.09983v2. [CrossRef]
14. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374v2.
15. Guo, T.; Gao, H. Content Enhanced BERT-based Text-to-SQL Generation. *arXiv* **2020**, arXiv:1910.07179v5.
16. Norouzi, S.; Tang, K.; Cao, Y. Code Generation from Natural Language with Less Prior and More Monolingual Data. *arXiv* **2021**, arXiv:2101.00259v2.
17. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.

18. Hellendoorn, V.J.; Sutton, C.; Singh, R.; Maniatis, P.; Bieber, D. Global relational models of source code. In Proceedings of the International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019.
19. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. Intellicode Compose: Code Generation Using Transformer. *arXiv* **2020**, arXiv:2005.08025.
20. Ahmad, W.U.; Chakraborty, S.; Ray, B.; Chang, K. Unified Pre-training for Program Understanding and Generation. *arXiv* **2021**, arXiv:2103.06333v2.
21. Phan, L.; Tran, H.; Le, D.; Nguyen, H.; Anibal, J.; Peltekian, A.; Ye, Y. CoTexT: Multi-task Learning with Code-Text Transformer. *arXiv* **2021**, arXiv:2105.08645v1.
22. Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv* **2021**, arXiv:2102.04664v2.
23. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkorei, J.; Jones, L.; Aidan; Gomez, L.; Kaiser, Ł.; Polosukhin, I. Attention Is All You Need. *arXiv* **2017**, arXiv:1706.03762v5.