The Journal of Systems and Software 214 (2024) 112066

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Context-aware code generation with synchronous bidirectional decoder ☆

Xiangyu Zhang [a], Yu Zhou [a,*], Guang Yang [a], Tingting Han [b], Taolue Chen [b]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] School of Computing and Mathematical Sciences, Birkbeck, University of London, UK

ARTICLE INFO

Keywords:
Code generation
Transition system
Bidirectional decoder
Neural network

ABSTRACT

Code generation aims to map natural language descriptions to code snippets. Recent approaches using sequence-to-tree models have shown promising results. However, they generally adopt an autoregressive way to predict the next token based on previous ones and do not consider potential future tokens. To address this issue, we propose Contextor, a novel context-sensitive model employing a bidirectional decoder to generate tokens in two different orders synchronously and interactively. Specifically, we employ two decoders to generate two sequences of different traversals and share their context knowledge via the attention mechanism. As a result, our model can synthesize both previous and future information simultaneously. To alleviate the information leakage problem caused by the teacher-forcing training strategy and bidirectional decoding, we propose an adapted scheduled sampling technique to prevent the decoders from contacting the actual label. Furthermore, Contextor also features a bidirectional beam search algorithm to better interact with both decoders. Experimental results demonstrate that our approach outperforms the state-of-the-art baselines.

1. Introduction

Neural code generation aims to map natural language descriptions to code snippets via deep learning, and has attracted extensive attention from natural language processing and software engineering communities. With the help of neural code generation tools, developers can accomplish their programming tasks more efficiently (Yang et al., 2023a; Balog et al., 2016).

In previous studies, researchers have employed models with sequence-to-sequence (seq2seq) architecture on the code generation task. Generally, seq2seq models generate the target sequences in an autoregressive way, inferring the current token based on previous tokens that the model has generated. These models are designed to mimic human intuition and the thought process involved in reading and writing code. They have shown to be highly effective in generating code for various tasks (Yin and Neubig, 2018; Beau and Crabbé, 2022; Sun et al., 2020; Yang et al., 2023b). However, the unidirectional dependence on sequences may lead to context inconsistency. The following example is from the CoNaLa (Yin et al., 2018) dataset, where the natural language description is *Get the first element of each row from a tensor A with two dimensions and return a list*. Unidirectional models may output `[i[0].item() for i in A.tolist()]`, as the model does not realize that it has taken out the value of the tensor *i[0]* and does not need to convert *tensor A* to *list*. In contrast, the correct code snippets are supposed to be either `[i[0] for i in A.tolist()]` or `[i[0].item() for i in A]`.

To address this issue, bidirectional models are leveraged which can infer the target sequences from both left to right and right to left. Such models utilize bidirectional decoding simultaneously and interactively (Zhang et al., 2020; Zhou et al., 2019). Compared with models based on the autoregressive decoding, the bidirectional decoder allows to take into account both left and right context of the target text when generating the output, which can lead to more accurate and coherent text generation. Furthermore, in an autoregressive model, errors can propagate through the generations, as each output is conditioned on the previous ones. In contrast, bidirectional decoders generate output in two directions, reducing the impact of errors on subsequent generation.

Despite that a bidirectional decoder can provide more contextual information, there are some potential disadvantages. For example, in natural language generation (NLG) tasks, the context information provided by left-to-right decoding may not be useful for right-to-left decoding because the information at both ends of a sentence may be logically unrelated. We observe that code generation tasks exhibit the same phenomenon. As illustrated in Table 1, we use UniXcoder (Guo et al., 2022) to encode Python code and extract attention scores to evaluate the relevance between tokens. Following the previous work (Kou

☆ Editor: Lingxiao Jiang.

* Corresponding author.
E-mail addresses: zhangx1angyu@nuaa.edu.cn (X. Zhang), zhouyu@nuaa.edu.cn (Y. Zhou), novelyg@outlook.com (G. Yang), t.han@bbk.ac.uk (T. Han), t.chen@bbk.ac.uk (T. Chen).

https://doi.org/10.1016/j.jss.2024.112066
Received 3 April 2023; Received in revised form 11 April 2024; Accepted 14 April 2024
Available online 16 April 2024

Table 1
Using UniXcoder to encode and extract attention scores from the CoNaLa dataset. Attention_{first} denotes the attention score of the first layer of UniXcoder, Attention_{last} represents the attention score of the last layer, and $\text{Attention}_{average}$ indicates the average score across all layers. *Average* denotes the average attention score of the current word towards other words excluding special tokens, i.e., $[CLS], [EOS]$.

	First vs. last	Second vs. penultimate	Third vs. antepenultimate
Attention_{first}	.011	.007	.021
Average	**.042**	**.037**	**.049**
Attention_{last}	.023	.027	.039
Average	**.059**	**.059**	**.063**
$\text{Attention}_{average}$	.007	.006	.017
Average	**.053**	**.052**	**.055**

et al., 2023), we employ three evaluation methods to calculate relevance, i.e., taking the attention scores from the first layer, the last layer, and averaging the scores across all layers. We also calculate the average attention score of the current word to all other words. We can find that, for programming languages, the relevance between tokens at both ends of the code is much lower than the average score. This suggests that the traditional bidirectional decoding approach struggles to provide effective contextual information. In addition, the information sharing mechanism, while incorporating contextual information, inevitably introduces noise to bidirectional models. Lastly, NLG models utilize the teacher-forcing strategy to train the model by inputting the previous ground-truth token to predict the next token. However, the bidirectional decoder consists of two decoders that share information. When training one decoder using the teacher-forcing strategy, the ground-truth tokens are leaked to the other decoder during information sharing (Zhang et al., 2020), which adversely affects the training effectiveness.

Unlike natural language, code contains stronger structural information that can be represented as a tree (Wang et al., 2022; Hussain et al., 2020; Jiang et al., 2021b) or graph (Guo et al., 2021). As a result, there is potential to integrate the structure information of code into bidirectional decoding models, enabling the model to capture the structural information besides other contextual aspects.

In this paper, we propose Contextor, a sequence-to-tree (seq2tree) model with a context-sensitive bidirectional decoder for code generation. Our model is designed to interact with context information at the code structure level by utilizing bidirectional decoding strategies based on Abstract Syntax Trees (ASTs) traversal order. Compared with vanilla bidirectional decoding, sharing context information based on the traversal order of the ASTs can better represent the structural correlation between ASTs' nodes. Furthermore, by modeling the context of the nodes, the model can also capture more grammatical information, thus generating code with higher grammatical correctness.

In addition to exploring the structural characteristics of the code, we propose a scheme to optimize the training and inference of bidirectional decoding. We introduce the Adaptive Scheduled Sampling mechanism to mitigate the issue of information leakage during training the bidirectional decoder. To improve the results of the bidirectional model, we also employ the bidirectional beam search algorithm, which allows the model to generate multiple candidate results and obtain better solutions during inference. The experimental results show that our proposed method can achieve competitive results with fewer model parameters. Moreover, our approach demonstrates the superiority over pre-training models in terms of code grammar and structural correctness.

In summary, the main contributions of the current paper are as follows.

- We propose Contextor that enhances context knowledge by using a synchronous bidirectional decoder to improve the contextual coherence for the code generation task. Additionally, Contextor features a novel sampling technique that can alleviate the issue of information leakage, and further improve the performance of the model.
- We conduct comprehensive experiments and evaluate the performance of our proposed approach Contextor on the public dataset, and the results show that Contextor outperforms the state-of-the-art baselines.

Structure of the paper. The remainder of the paper is organized as follows. In Section 2, we provide a brief introduction to the background of this study. Section 3 describes the details of our approach. In Section 4, we present the experimental settings and comparative results on the public dataset. Sections 5 and 6 discuss threats to validity and related works, respectively. Finally, in Section 7, we conclude our study and outline future research directions.

To facilitate the replication and reuse of Contextor, we make our source code, trained models, as well as the datasets in the GitHub repository publicly available.[1]

2. Background

2.1. Seq2Tree techniques

In this part, we mainly introduce ASDL (Zephyr Abstract Syntax Description Language), a context-free grammar used to describe the structure of AST (Wang et al., 1997). In addition, we also briefly introduce two state-of-the-art seq2tree code generation methods.

ASDL. ASDL is used to define the abstract syntax of compiler intermediate representations (IRs) and other tree-like data structures. Similar to how regular expressions and context-free grammars describe the lexical and syntactic structures of programming languages, ASDL offers a concise way of representing the abstract syntax of programming languages. ASDL grammar consists of two main constructs: *types* and *constructors*. As shown in Fig. 1, *stmt* and *expr* are examples of *composite types*, which represent Python statements and expressions, respectively. *composite types* are defined by a series of *constructors*, each of which has fields that define its subtrees. For instance, a *FunctionDef constructor* has six fields, each of which specifies the structure of its corresponding subtree. Each field is assigned a cardinality to indicate the number of values it holds (single, optional ? and sequential *). ASDL provides a concise notation for describing the abstract syntax of programming languages, which is leveraged in this work.

TRANX. TRANX (Yin and Neubig, 2018) is a transition-based neural semantic parser, aiming to generate ASTs (rather than code snippets directly). Since tree-structure data are harder be directly put into deep learning models, TRANX applies a transition system based on ASDL to describe ASTs.

This system provides three actions to decompose the generation procedure of an AST into a sequence of tree-constructing actions: (1) **APPLYCONSTR [c]** action applies a constructor c to the composite frontier field with the same type of c. This action appends the constructor to a list of constructors if the frontier field has a sequential cardinality. (2) **REDUCE** action marks that the generation for a field with optional (?) or sequential (*) has already been completed. (3) **GENTOEKN [v]** action expands a primitive frontier field to generate a token v.

Specifically, given an input of NL utterance x, instead of mapping x to code snippet y directly, TRANX employs the transition system to convert x to an intermediate representation z, which is guided by the user-defined, domain-specific grammar specified under the ASDL formalism. Then, it converts z to code by rules. The probability of z can be defined as:

$$p(z \mid x) = \prod_t p\left(a_t \mid a_{t-1}, x\right)$$

[1] https://github.com/NUAAZXY/CONTEXTOR.

```
stmt = FunctionDef ( identifier name, arguments args,
                     stmt* body, expr* decorator_list,
                     expr? returns, string? type_comment)
     | Assign ( expr* targets, expr value,
                string? type_comment)
     | Expr ( expr value)
```

```
expr = Compare ( expr left, compop* ops,
                 expr* comparators)
     | Call ( expr func, expr* args, keyword* keywords)
     | Attribute(expr value, identifier attr,
                 expr_context ctx)
```

Fig. 1. An example of ASDL.

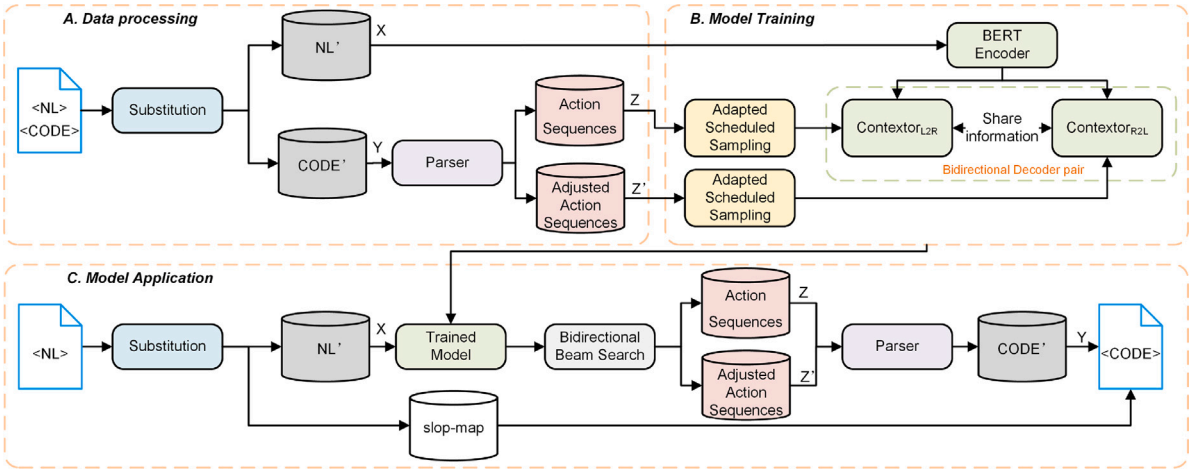


Fig. 2. The workflow of Contextor.

where a_t is the attention representing the correlation between NL utterance x and the current hidden state generated by the decoder.

TRANX uses an encoder–decoder framework with a standard bidirectional LSTM encoder and a unidirectional LSTM decoder. The encoder encodes x into vectorial representation then the decoder decodes the output attention a_t through the attention mechanism and maps it to an action. This approach gives the parser high accuracy and generalizability. The accuracy of TRANX is achieved through its use of information from the syntax of the target meaning representations (MR), which allows it to constrain the output space and model the flow of information in the sentence. Meanwhile, its high generalizability makes it easy to apply to new types of MR by simply writing a new abstract syntax description that corresponds to the allowable structures in the MR.

BertranX. BertranX (Beau and Crabbé, 2022) is an extension of the TRANX system that employs BERT (Devlin et al., 2019) as an encoder to better understand the input NL utterances. Additionally, BertranX adapts the transition system introduced in TRANX by using Earley Parser (Earley, 1970) to describe ASTs by a stack of dotted rules. The Earley Parser is an algorithm for parsing strings that belong to a given context-free language. In the case of BertranX, the Earley Parser is employed to parse the ASTs. For instance, $A \rightarrow \alpha \bullet X\beta$ is a dotted rule, where α represents a sequence of grammar symbols whose subtrees have already been generated and $X\beta$ is a sequence of grammar symbols that are yet to be generated.

Similar to TRANX, BertranX also defines actions to describe ASTs: (1) **PREDICT(C)** action aims to start the generation of a new subtree. The GENERATE action adds a new node to a tree, whereas a COMPLETE action signals the end of subtree generation and allows the process to continue with the parent node. The set of PREDICT(C) is equivalent to the number of constructors in the ASDL grammar, as each action is parameterized the ASDL rule constructor (C). These constructors are necessary for producing concrete ASTs from derivations. CLOSE action is used to stop the generation if all rules are completed, and this action is covered by PREDICT(C). (2) **GENERATE(V)** action is responsible for generating terminal or primitive symbols. In the Python ASDL grammar, ASTs are created with primitive leaf types (e.g., identifier, int, string, constant) that require actual values. The set of possible values V for these primitives is infinite, which means that the set of GENERATE actions is also infinite.

2.2. Synchronous bidirectional decoding

Autoregressive models have limitations in predicting the next token as they only rely on the previous tokens and do not have access to future information. In contrast, synchronous bidirectional decoding is a technique that leverages both past and future information simultaneously to improve prediction accuracy.

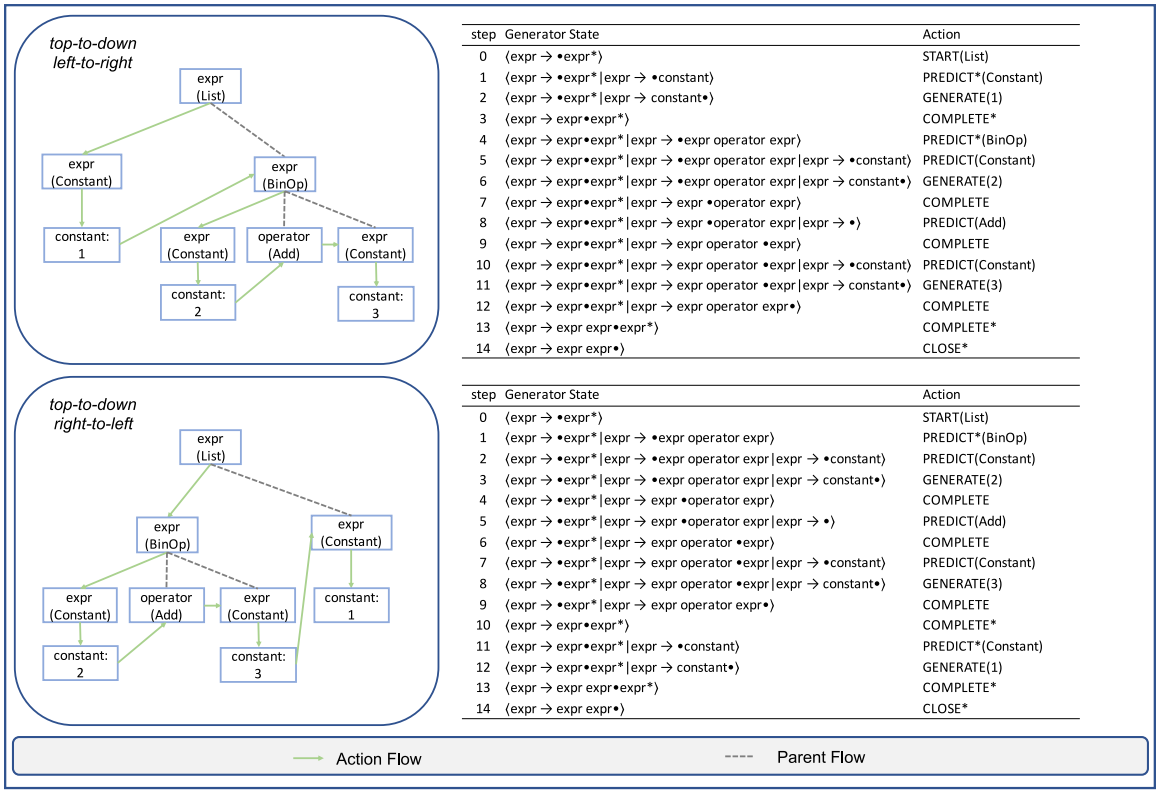
Previous studies (Zhang et al., 2020; Zhou et al., 2019) employ a bidirectional decoder to generate two sequences, one from left to right and the other from right to left. Each decoder uses both previous and future predictions synchronously to generate context-coherent information, which can be expressed as:

$$p(y \mid x) = \begin{cases} \prod_t p\left(\overrightarrow{y_t} \mid \overrightarrow{y_0}, \ \ldots, \ \overrightarrow{y_{t-1}}, x, \overleftarrow{y_0}, \ldots, \overleftarrow{y_{t-1}}\right) & \text{if left-to-right} \\ \prod_t p\left(\overleftarrow{y_t} \mid \overleftarrow{y_0}, \ \ldots, \ \overleftarrow{y_{t-1}}, x, \overrightarrow{y_0}, \ldots, \overrightarrow{y_{t-1}}\right) & \text{if right-to-left} \end{cases} \quad (1)$$

where $\overrightarrow{y_t}$ (resp. $\overleftarrow{y_t}$) is the current output of the left-to-right (resp. right-to-left) decoder, and x is the input of the model. By considering the surrounding context of each token, these bidirectional decoders are able to generate more accurate predictions and improve the overall performance of the model.

3. Approach

The workflow of Contextor is shown in Fig. 2, which consists of three main components, i.e., *data processing*, *model training*, and *model application*. In *data processing*, we normalize variables and constant names using a substitution method. To fully utilize contextual information, we then parse code snippets into two sequences of predefined actions. These sequences differ in that one is generated from a top-down, left-to-right traversal (L2R) of the AST, while the other is generated from a top-down, right-to-left (R2L) traversal. In *model training*, we feed both action sequences (denoted as Y) and adjusted



step	Generator State	Action
0	⟨expr → •expr*⟩	START(List)
1	⟨expr → •expr*|expr → •constant⟩	PREDICT*(Constant)
2	⟨expr → •expr*|expr → constant•⟩	GENERATE(1)
3	⟨expr → expr•expr*⟩	COMPLETE*
4	⟨expr → expr•expr*|expr → •expr operator expr⟩	PREDICT*(BinOp)
5	⟨expr → expr•expr*|expr → •expr operator expr|expr → •constant⟩	PREDICT(Constant)
6	⟨expr → expr•expr*|expr → •expr operator expr|expr → constant•⟩	GENERATE(2)
7	⟨expr → expr•expr*|expr → expr •operator expr⟩	COMPLETE
8	⟨expr → expr•expr*|expr → expr •operator expr|expr → •⟩	PREDICT(Add)
9	⟨expr → expr•expr*|expr → expr operator •expr⟩	COMPLETE
10	⟨expr → expr•expr*|expr → expr operator •expr|expr → •constant⟩	PREDICT(Constant)
11	⟨expr → expr•expr*|expr → expr operator •expr|expr → constant•⟩	GENERATE(3)
12	⟨expr → expr•expr*|expr → expr operator expr•⟩	COMPLETE
13	⟨expr → expr expr•expr*⟩	COMPLETE*
14	⟨expr → expr expr•⟩	CLOSE*

step	Generator State	Action
0	⟨expr → •expr*⟩	START(List)
1	⟨expr → •expr*|expr → •expr operator expr⟩	PREDICT*(BinOp)
2	⟨expr → •expr*|expr → •expr operator expr|expr → •constant⟩	PREDICT(Constant)
3	⟨expr → •expr*|expr → •expr operator expr|expr → constant•⟩	GENERATE(2)
4	⟨expr → •expr*|expr → expr •operator expr⟩	COMPLETE
5	⟨expr → •expr*|expr → expr •operator expr|expr → •⟩	PREDICT(Add)
6	⟨expr → •expr*|expr → expr operator •expr⟩	COMPLETE
7	⟨expr → •expr*|expr → expr operator •expr|expr → •constant⟩	PREDICT(Constant)
8	⟨expr → •expr*|expr → expr operator •expr|expr → constant•⟩	GENERATE(3)
9	⟨expr → •expr*|expr → expr operator expr•⟩	COMPLETE
10	⟨expr → expr•expr*⟩	COMPLETE*
11	⟨expr → expr•expr*|expr → •constant⟩	PREDICT*(Constant)
12	⟨expr → expr•expr*|expr → constant•⟩	GENERATE(1)
13	⟨expr → expr expr•expr*⟩	COMPLETE*
14	⟨expr → expr expr•⟩	CLOSE*

Fig. 3. The procedure of parsing code snippets.

action sequences (denoted as Y') into the bidirectional decoder to train the model. Training a bidirectional decoder using teacher-forcing may result in the issue of information leakage. To solve this, rather than use the teacher-forcing decoding mechanism, we employ a novel sampling method to alleviate the problem of information leakage during training. During *model application*, We use bidirectional beam search algorithm to optimize and utilize the decoding results from both ends, thus enhancing the generation capacity.

In the rest of this section, we present the details of the preprocessing (Section 3.1), the model structure (Section 3.2), the bidirectional beam search algorithm (Section 3.3) as well as the adapted scheduled sampling technique (Section 3.4).

3.1. Data processing

Substitution. Programming languages have an infinite set of lexical symbols that include a wide range of variable names and constant identifiers. Instead of learning the statistics of a particular set of symbols, we utilize a preprocessing method to normalize variable and constant names. The goal of preprocessing is to replace the actual name of a variable with a set of predefined standardized names that are known to the statistical model. This replacement step renames all variables in both natural language and code using a common name. Specifically, we rename all variables defined by programmers as *var_0, var_1, etc.*, and all lists as *lst_0, lst_1, etc.*.

Transition system. The conventional seq2tree model generates AST nodes in a top-down, left-to-right traversal order. In our experiment, we generate code snippets based on the two different traversal orders of the AST, and share their context information synchronously. As illustrated in Fig. 3, we utilize the parser proposed in BertranX (Beau and Crabbé, 2022) to traverse the abstract syntax tree of the code `[1, 2+3]` in two distinct directions. The first traversal sequence follows the top-down, left-to-right traversal order as previous methods. To ensure that the model can capture more structured information, we adopt a top-down, right-to-left traversal method in the second sequence. Specifically, we first perform an up-to-down traversal of the AST. If the current node has more than one child, we reverse its children with their subtrees, from N1, N2, N3 to N3, N2, N1, and so on. Next, we traverse the adjusted tree in top-down, left-to-right traversal order. By integrating contextual information from two directions, our bidirectional decoder can utilize both context and structural knowledge simultaneously.

3.2. Model architecture

The architecture of our proposed model is depicted in Fig. 4. Our model contains a BERT encoder and a context-aware bidirectional decoder. We will provide two decoder schemes, namely bidirectional LSTM decoder and bidirectional Transformer decoder, which will be elaborated in this section.

Encoder. We use BERT (Devlin et al., 2019) as the encoder of our seq2tree model. BERT employs a transformer encoder, which makes it capable of bidirectional encoding and feature extraction (Yang et al., 2021). Given an NL sequence $x = (x_0, x_1, \ldots, x_{m-1})$, BERT outputs an intermediate hidden state which represents the meaning of the NL, i.e., $h^{(enc\_out)} = BERT(x)$.

Bidirectional LSTM Decoder. First, we introduce CONTEXTOR using the bidirectional LSTM decoder, i.e. Contextor_{LSTM}. As shown in Fig. 3, an AST contains both semantic and structural information of code, which introduces specific syntax information to the model. However, traditional neural network models have difficulty in leveraging ASTs (Zhang et al., 2023). Unidirectional models serialize the nodes of ASTs, such as TRANX (Yin and Neubig, 2018). This serialization method is compatible with traditional generative models, but cannot establish effective contextual connections. Although GrammarCNN (Sun et al., 2019) and CGML (Xie et al., 2021) consider the connections

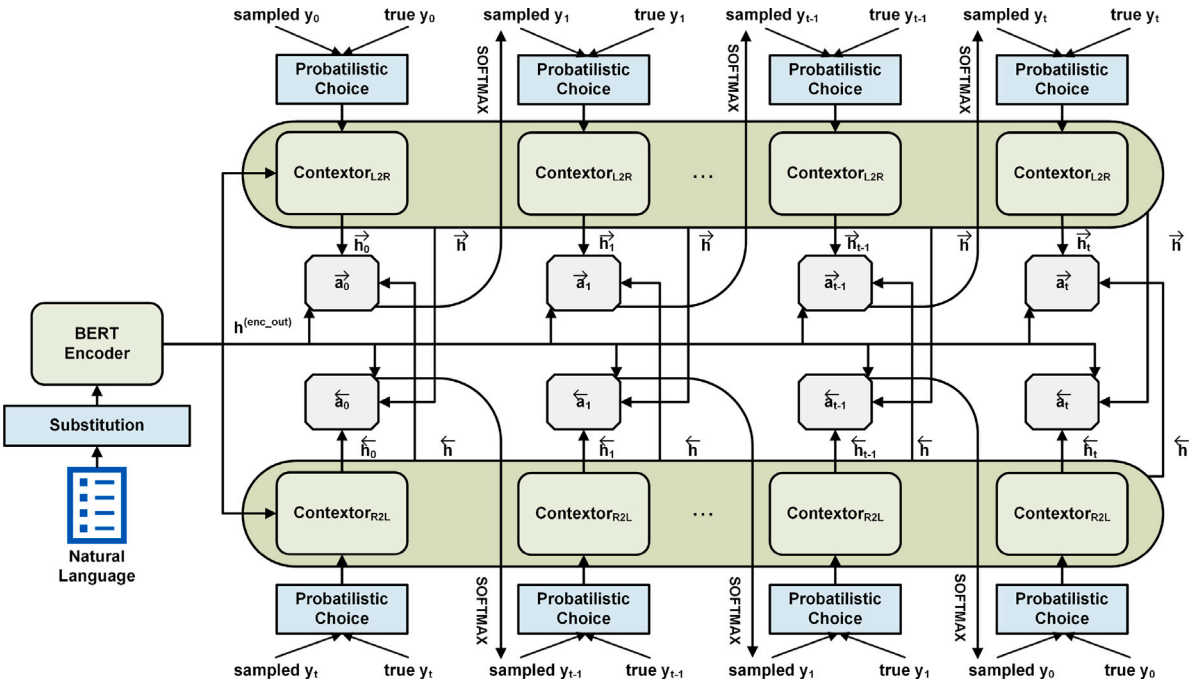


Fig. 4. The Contextor model architecture.

between contexts, these models are essentially unidirectional models and cannot measure the structural information of code from a global perspective.

In our approach, we use a bidirectional decoder consisting of two LSTM decoders, namely LSTM_{L2R} and LSTM_{R2L}, to decode sequences of actions in different orders. The probability of the output action sequence z given the input sequence x is defined as

$$p(z|x) = \begin{cases} \prod_t p\left(\vec{z}_t \mid \vec{z}_0, \ \ldots, \vec{z}_{t-1}, h^{(enc\_out)}, \overleftarrow{z}_0, \ldots, \overleftarrow{z}_{t-1}\right), \\ \qquad \text{if left-to-right traversal} \\ \prod_t p\left(\overleftarrow{z}_t \mid \overleftarrow{z}_0, \ \ldots, \overleftarrow{z}_{t-1}, h^{(enc\_out)}, \vec{z}_0, \ldots, \vec{z}_{t-1}\right), \\ \qquad \text{if right-to-left traversal} \end{cases}$$

where $\vec{z}_t$ denotes the hidden state of the LSTM_{L2R} at time step t, and $\overleftarrow{z}_t$ represents the LSTM_{R2L}'s output at time step t.

Take the LSTM_{L2R} as an example. As shown in Fig. 4, we employ the attention mechanism to share the context information of the decoders. At time step t, we input the guiding label $\vec{y}_{t-1}$, the previous attention $\vec{a}_{t-1}$, and the previous hidden state $\vec{h}_{t-1}$ into the LSTM decoder and obtain a new hidden state $\vec{h}_t = LSTM\left(\left[\vec{y}_{t-1} : \vec{a}_{t-1}\right], \vec{h}_{t-1}\right)$ where $\vec{y}_{t-1}$ is the previous token, $\vec{a}_{t-1}$ is the global attention output at time step $t-1$, $[:]$ denotes concatenation of two vectors. In particular, at time step 0, after the encoder outputs an intermediate hidden state $h^{(enc\_out)}$, the decoder outputs the first hidden state $\vec{h}_0$ by $\vec{h}_0 = LSTM\left(\vec{0}, h^{(enc\_out)}\right)$. where $\vec{0}$ has the same dimension as the input to the LSTM cell.

The global attention mechanism, denoted by $\vec{a}_t$, is comprised of two fundamental types of attention, specifically the encoder–decoder attention $\vec{a}_t^{(enc)}$ and the decoder–decoder attention $\vec{a}_t^{(dec)}$. The encoder–decoder attention $\vec{a}_t^{(enc)}$ computes the correlation between the encoder outputs and the current hidden state generated by the decoder. This attention mechanism computes a weighted sum of the encoder-side context vectors and can be expressed using $\vec{a}_t^{(enc)} = \sum_{i=0}^{m-1} w_{ti}^{(enc)} h_i^{(enc\_out)}$ where

$$w_{ti}^{(enc)} = \frac{exp\left(\vec{m}_{ti}\right)}{\sum_{j=0}^{m-1} exp\left(\vec{m}_{tj}\right)}, \quad \vec{m}_{ti} = \mathbf{V}^{(enc)} \tanh\left(\mathbf{W}^{(enc)} \vec{h}_t + \mathbf{U}^{(enc)} h_i^{(enc\_out)}\right)$$

Here, **V**, **W**, and **U** are three neural networks with mutually independent parameters.

The above formulas compute the relevance between the input NL and the current hidden state generated by the LSTM_{L2R}. In addition, the Contextor_{LSTM} also focuses on the connection between the two decoders. We propose a decoder–decoder attention mechanism to calculate the relevance between the current LSTM_{L2R}'s hidden state and the hidden states of LSTM_{R2L}. We then incorporate future knowledge into the current decoding state through a weighted sum, thus making the model context-sensitive. The decoder–decoder attention, denoted as $\vec{a}_t^{(dec)}$, can be expressed as $\vec{a}_t^{(dec)} = \sum_{k=0}^{t} w_{tk}^{(dec)} \overleftarrow{h}_k$ where

$$w_{tk}^{(dec)} = \frac{exp\left(\vec{n}_{tk}\right)}{\sum_{j=0}^{t} exp\left(\vec{n}_{tj}\right)} \text{ and } \vec{n}_{tk} = \mathbf{V}^{(dec)} \tanh\left(\mathbf{W}^{(dec)} \vec{h}_t + \mathbf{U}^{(dec)} \overleftarrow{h}_k\right)$$

Note $\overleftarrow{h}$ restores the past hidden states that the LSTM_{R2L} has decoded.

As future knowledge is obtained through weighting, this information sharing method inevitably introduces noise to the model. To reduce the impact of noise introduced by bidirectional decoding, we developed a lightweight noise masking network. This network automatically sets attention weights that are irrelevant to the current generation to zero, effectively mitigating the impact of noise. The noise masking network can be described as follows:

$$\vec{w}_{tk}^{(dec)} = \begin{cases} \vec{w}_{tk}^{(dec)}, & \vec{w}_{tk}^{(dec)} \geq \text{sigmoid}\left(W^{(noise)}\left[\vec{h}_t : \overleftarrow{h}_k\right]\right) \\ 0, & \vec{w}_{tk}^{(dec)} < \text{sigmoid}\left(W^{(noise)}\left[\vec{h}_t : \overleftarrow{h}_k\right]\right) \end{cases}$$

As described earlier, we utilize the additive attention mechanism to obtain two kinds of basic attention $\vec{a}_t^{(enc)}$ and $\vec{a}_t^{(dec)}$. The first attention,

$\vec{a}_t^{(enc)}$, is used to capture the correlation between the current hidden state $\overleftarrow{h}_t$ and the input natural language sequence. This allows the model to focus more on the significant tokens in the input sequence. The second attention, $\vec{a}_t^{(dec)}$, measures the correlation between the current hidden state $\vec{h}_t$ and the history hidden states $\overleftarrow{h}$ generated by the other decoder. By doing so, we can extract future information and utilize it to generate a more informed and contextual output.

We combine both types of attention scores, to obtain the attention $\vec{a}_t = \tanh\left(\mathbf{W}\left[\vec{h}_t : \vec{a}_t^{(enc)} : \vec{a}_t^{(dec)}\right]\right)$. As a result, the final probability of the result can be expressed as $p\left(z_t \mid x\right) = \text{softmax}\left(\mathbf{V}^{(res)}\vec{a}_t\right)$.

Note that our model has two types of output (primitives and constructors), which correspond to different embeddings (as shown in Section 2.1). Therefore, we employ two generators to generate each type of action. The probability of the target action can be defined as:

$$p\left(\vec{z}_t \middle| x\right) = \begin{cases} p\left(\vec{z}_t = GENERATE(v) \middle| x\right) & = \text{softmax}\left(\mathbf{V}^{(pmt)}\vec{a}_t\right), \\ \text{if primitives} & \\ p\left(\vec{z}_t = PREDICT(c) \middle| x\right) & = \text{softmax}\left(\mathbf{V}^{(cst)}\vec{a}_t\right), \\ \text{if constructors} & \end{cases}$$

Similarly, the LSTM_{R2L} outputs can be defined as follow:

$$p\left(\overleftarrow{z}_t \middle| x\right) = \begin{cases} p\left(\overleftarrow{z}_t = GENERATE(v) \middle| x\right) & = \text{softmax}\left(\mathbf{V}^{(pmt)}\overleftarrow{a}_t\right), \\ \text{if primitives} & \\ p\left(\overleftarrow{z}_t = PREDICT(c) \middle| x\right) & = \text{softmax}\left(\mathbf{V}^{(cst)}\overleftarrow{a}_t\right), \\ \text{if constructors} & \end{cases}$$

Bidirectional Transformer Decoder. Transformer (Vaswani et al., 2017a) can process the entire input data at once, unlike sequential data processing methods such as LSTM. This characteristic allows for parallel computing and considerably reduces processing time, resulting in faster model training. In this paper, we also introduce a method that utilizes bidirectional Transformer (Zhou et al., 2019) as the decoder, i.e. $\text{Contextor}_{Transformer}$. Due to the fact that the Transformer is based on self-attention mechanism and cannot capture contextual hidden states at every time step during training, we designed a context-sharing mechanism based on self-attention.

The vanilla Transformer uses self-attention to model the context, and its hidden states can be represented as $\vec{a}_t^{(self)} = \sum_{i=0}^{t} w_{ti}\vec{V}_i$ where

$$w_{ti}^{(self)} = \frac{exp\left(\vec{m}_{ti}\right)}{\sum_{j=0}^{t} exp\left(\vec{m}_{tj}\right)} \text{ and } \vec{m}_{ti} = \frac{\vec{Q}_t\vec{K}_i^T}{\sqrt{d_k}}$$

Here, Q, K, and V represent the query, key, and value vectors introduced by the Transformer.

Due to the parallel computing method of the Transformer, we are unable to exchange the hidden states of two decoders sequentially. As an alternative, we share information by exchanging the K and V vectors between the decoders. Specifically, we use the Q from Transformer_{L2R} as the query vector and the K and V from Transformer_{R2L} as the key and value vectors, respectively. The hidden state can be represented as $\vec{a}_t^{(context)} = \sum_{i=0}^{t} w_{ti}\overleftarrow{V}_i$ where

$$w_{ti}^{(context)} = \frac{exp\left(\vec{n}_{ti}\right)}{\sum_{j=0}^{t} exp\left(\vec{n}_{tj}\right)} \text{ and } \vec{n}_{ti} = \frac{\vec{Q}_t\overleftarrow{K}_i^T}{\sqrt{d_k}}$$

The attention output can be represented as the fusion of $\vec{a}_t^{(self)}$ and $\vec{a}_t^{(context)}$, denoted as $\vec{a}_t = \vec{a}_t^{(self)} + \theta \times \tanh(\mathbf{W}^{(proj)}\vec{a}_t^{(context)})$, where θ is the hyperparameter controlling attention towards future knowledge. Similarly, the output of Transformer_{R2L} can be represented as $\overleftarrow{a}_t = \overleftarrow{a}_t^{(self)} + \theta \times \tanh(\mathbf{W}^{(proj)}\overleftarrow{a}_t^{(context)})$

By incorporating bidirectional propagation, Contextor is able to model dependencies between past and future node pairs, which enhances its capacity for contextual analysis. In contrast to models with unidirectional dependencies, bidirectional dependency relationships offer Contextor more comprehensive connections. This enhanced connectivity empowers the model with sufficient contextual and structural knowledge of the abstract syntax tree.

3.3. Bidirectional beam search

To enhance the quality of code generation by facilitating the production of multiple candidate outputs, we utilize the beam search algorithm for decoding at inference. The beam search algorithm can efficiently find the optimal solution in a limited search space, producing a solution close to the optimal solution in the entire search space. By setting a beam size k, we reserve the top k temporary paths with the highest probability at each time step. In a unidirectional decoder, the beam search algorithm only needs to generate multiple candidate results based on the probability distribution of the current decoder output. However, a bidirectional decoder comprises two unidirectional decoders that share information. During the training phase, we combine the outputs of both decoders and optimize the model at each time step based on the probability distribution generated by the current time step. However, at inference, each decoder needs to combine the k-best temporary paths preserved by the beam search algorithm with k-best contextual information generated by the other decoder. The complexity of combining each path with k contextual information is significantly high. Hence, we need select the most accurate result from the candidate outputs as the contextual information.

To better utilize context knowledge from the decoders and reduce time complexity, we propose a simple Bidirectional Beam Search algorithm. Instead of enumerating all the paths generated by the other decoder when calculating $\vec{a}_t$, we choose the path with the current highest probability score as the external knowledge. Fig. 5 shows an example of a bidirectional beam search for generating code `list(t)`. At time step t_6, we obtain two temporary paths using the beam search algorithm in the L2R Decoder. When predicting the next token, we use the information of the highest-scoring path (indicated by the orange arrow) in the R2L Decoder as the contextual information for the current predicted token. In this way, we can reduce the time complexity and still benefit from the context knowledge of the other decoder.

3.4. Adapted scheduled sampling

Although bidirectional decoding can enable the model to absorb both past and future knowledge, it may cause information leakage problem when using the teacher-forcing training strategy. The teacher-forcing strategy predicts the next token $\vec{y}_t^{(predict)}$ instructed by a previous gold token $\vec{y}_{t-1}^{(gold)}$. For instance, as is shown in Fig. 6, at time step t_5, the L2R Decoder predicts node *cat* guided by the previous true token *torch* via teacher-forcing. However, the decoding information of R2L Decoder is also visible, which leads to the leakage of the true labels to L2R Decoder. An immediate consequence of information leakage is that the training loss in bidirectional decoding is misleadingly lower than that of the loss in unidirectional decoding. Information leakage may have a significant impact on inference performance. Due to the exposure of real labels during the training stage, both decoders will excessively rely on each other's information. If one decoder decodes an incorrect token during the inference stage, the other decoder will also be affected by it.

To alleviate information leakage caused by the bidirectional decoder and teacher forcing training strategy, previous work (Zhang et al., 2020) trains their decoders independently in the first training pass with teacher-forcing. Then, in the second training pass, they predict the

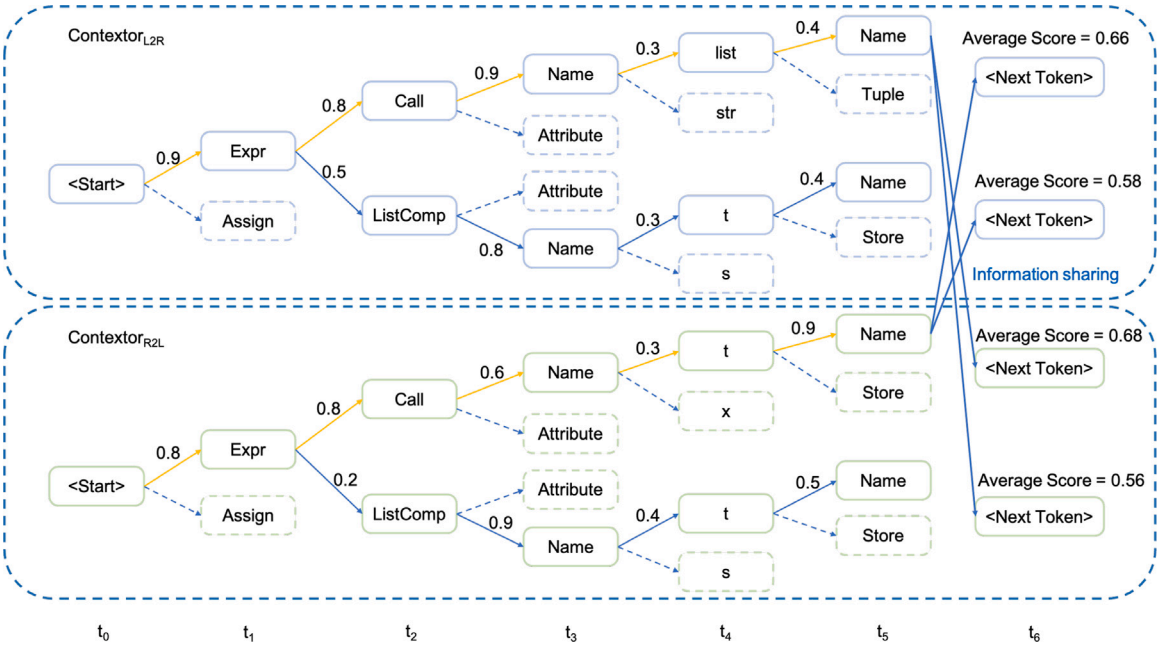


Fig. 5. Bidirectional beam search.

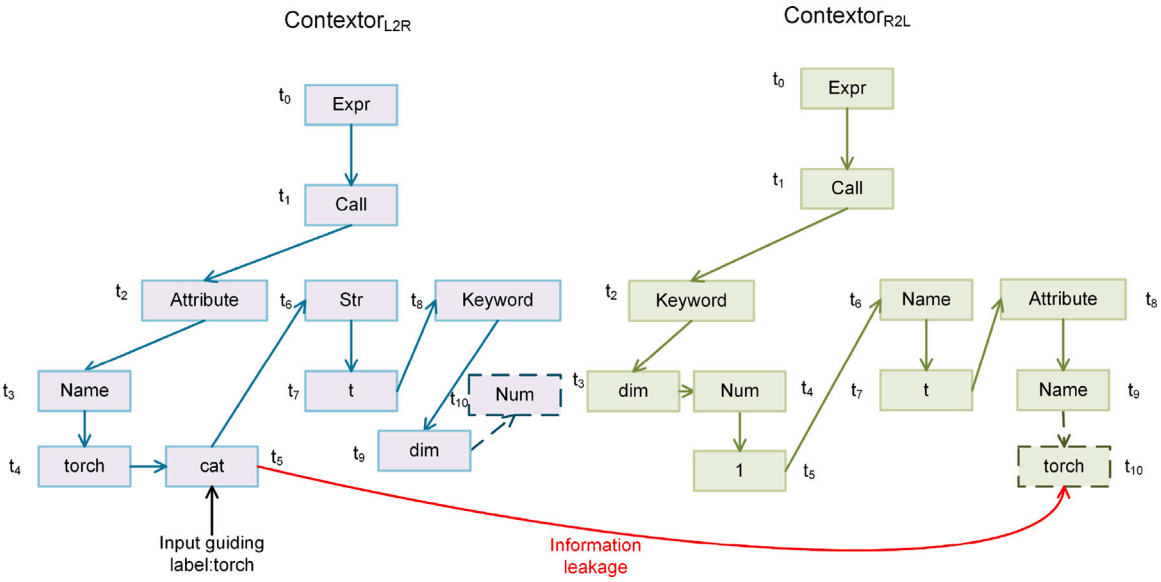


Fig. 6. The procedure of bidirectional decoding. Solid arrows represent the workflow of the decoder. Dashed arrows show the node generated at the next time step.

next token $\overrightarrow{y}_t^{(predict)}$ instructed by the previous predicted token $\overrightarrow{y}_{t-1}^{(predict)}$ which is called student-forcing strategy.

Although this method can somewhat alleviate the information leakage problem, the change from teacher-forcing to student-forcing is too sharp that the bidirectional models cannot adapt well. To bridge the gap between teacher-forcing to student-forcing, We employ scheduled sampling (Bengio et al., 2015) in our model. Scheduled sampling is used to solve the inconsistency between the distribution of input data during training and generation. In the early stage of training, this method mainly uses real tokens as the input of the decoder, which can quickly converge the model from a randomly initialized state to a relatively good state. As the training progresses, this method will gradually use more generated elements as the input of the decoder to solve the problem of inconsistent data distribution.

Their work can also be defined as a method to narrow the gap between teacher-forcing and student-forcing and is suitable to solve our problem. As shown in Fig. 4, We use the probability based on an exponential decay function to choose whether to use actual or predicted tokens. The exponential decay function can be defined as $prob_e = \lambda_1^e$

where e means the number of current training epochs, $prob_e$ is the probability of using the actual labels, λ_1 ($\lambda_1 < 1$) is a hyper-parameter that depends on the expected speed of convergence.

Note that the nodes generated earlier are more likely to be leaked. So we adapt the formula as

$$prob_{et} = \lambda_1^e + \left(\frac{\lambda_2}{maxstep}\right) * t - \lambda_2$$

where t is the time step of the generation. The term λ_1^e ensures that the model starts by using actual labels with high probability at the beginning of training, while the term $\left(\frac{\lambda_2}{maxstep}\right) * t - \lambda_2$ gradually increases the probability of using predicted tokens as training progresses because the tokens in earlier time steps are likely to be leaked while generating a

code snippet	natural language
pd.merge(df1, df2, on=['A', 'B'], how='inner')	Find all rows in Dataframe 'df2' that are also present in Dataframe 'df1', for the columns 'A','B'.
sorted(a, key=lamba x: (sum(x[1:3]), x[0]), reverse=True)	Sorting a list of tuples by the addition of second and third element of the tuple.

Fig. 7. Examples of CoNaLa.

Table 2
Hyper-parameters for model Contextor$_{LSTM}$.

Category	Hyper-parameter	Value
Model structure	Hidden size	512
	Attention size	512
	Dropout	0.3
Decay parameters	λ_1	0.95
	λ_2	0.2
Model training	Optimizer	Adam
	Learning rate	1e−4
	Batch size	32
Model application	Beam size	15

Table 3
Hyper-parameters for model Contextor$_{Transformer}$.

Category	Hyper-parameter	Value
Model structure	Hidden size	768
	Attention size	768
	Head num	12
	Layer num	12
	Dropout	0.1
Combining parameter	θ	0.2
Decay parameters	λ_1	0.95
	λ_2	0.2
Model training	Optimizer	AdamW
	Learning rate	5e−5
	Batch size	32
Model application	Beam size	20

sequence at training. By using this method, the model can either reduce information leakage or take advantage of teacher-forcing strategy to achieve faster convergence, depending on the specific situation.

4. Experiments

We mainly focus on the following four research questions:

RQ1: How effective is Contextor compared with state-of-the-art baselines?
RQ2: How does bidirectional decoding mechanism facilitate the generation of context-coherent code?
RQ3: What is the role of adapted scheduled sampling?
RQ4: What is the impact of different NL lengths, sequence lengths and code lengths on the performance of Contextor?

In RQ1, we compare our proposed method Contextor with three state-of-the-art baselines, i.e., (1) **TRANX**, which is a transition-based neural abstract syntax parser for semantic parsing and code generation. (2) **BertranX**, which is a state-of-the-art architecture that relies on a BERT encoder and a grammar-based decoder for code generation. (3) **UniXcoder** (Guo et al., 2022), a unified cross-modal pre-trained encoder for code representation. (4) **CodeGPT** (Lu et al., 2021b), a pre-trained decoder for code completion and text-to-code generation tasks. (5) **CodeT5+** (Wang et al., 2023), an open code pre-trained model for code understanding and generation. To have a fair comparison, we reuse the training parameters provided in the baselines.

In RQ2 and RQ3, we focus on evaluating the critical components of our work. Finally, in RQ4, we analyze the impact of NL lengths, sequence lengths and code lengths on the performance of Contextor and compare it with the baselines.

Dataset. We evaluate Contextor on the CoNaLa dataset (Yin et al., 2018), which consists of 600k NL-code pairs mined from *StackOverflow*, with 2879 of them having been manually rewritten. As shown in Fig. 7, CoNaLa includes annotated NL questions and their corresponding Python3 solutions. Although the code snippets in the CoNaLa dataset are at the line granularity, their complex and extensive code composition poses significant challenges to deep learning models.

Parameters. The hyper-parameters of Contextor$_{LSTM}$ and Contextor$_{Transformer}$ are shown in Tables 2 and 3 separately.

All the experiments are run on a workstation with Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz, 32 RAM, and a GeForce RTX2080Ti GPU with 11 GB memory. The running OS platform is Linux OS.

Table 4
Comparison between Contextor and baselines for CoNaLa datasets.

Approach	BLEU	CodeBLEU
TRANX	27.18	28.04
BertranX	30.02	30.34
UniXcoder	31.37	31.89
CodeGPT	35.27	33.13
CodeT5+	**35.92**	33.95
Contextor$_{LSTM}$	32.92	33.36
Contextor$_{Transformer}$	33.74	**34.80**
TRANX + mined	28.02	28.56
BertranX + mined	34.18	36.00
UniXcoder + mined	34.93	35.03
CodeGPT + mined	35.37	34.81
CodeT5+ + mined	36.21	35.12
Contextor$_{LSTM}$ + mined	35.92	37.02
Contextor$_{Transformer}$ + mined	**36.96**	**37.88**

4.1. Results

RQ1: How effective is contextor compared with state-of-the-art baselines? We use BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020) as automatic performance metrics. BLEU calculates the frequency of repeated words between the predicted sequence and the target one, which reflect the similarity of the two sentences. Although BLEU is the dominant metric in natural language NLG tasks such as neural machine translation (Wu et al., 2016; Stahlberg, 2020; Gu et al., 2017), which estimates the similarity between the predicted sequence and the target one by calculating the frequency of repeated words. However, it is insufficient to evaluate the quality of code snippets as even a single wrong token can make the code uncompilable. To address this, we also use CodeBLEU, which is a weighted sum of four parts as follows:

$$CodeBLEU = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{weight} + \gamma \cdot \text{Match}_{ast} + \delta \cdot \text{Match}_{df}$$

where BLEU$_{weight}$ is the weighted n-gram match which attaches more weight to keywords in the programming language, Match$_{ast}$ calculates the syntactic information by matching the tree structure, Match$_{df}$ is the semantic dataflow match.

We compared our method with five baseline programs in detail, and the experimental results are shown in Table 4. Contextor$_{Transformer}$ outperforms all the baselines in terms of the BLEU and CodeBLEU metric. In addition, the lightweight Contextor$_{LSTM}$ can achieve competitive

Table 5
Scores in every part of CodeBLEU on CoNaLa with mined data.

Model	BLEU	$BLEU_{weight}$	$Match_{ast}$	$Match_{df}$
TRANX	28.02	27.94	26.36	30.68
BertranX	34.18	33.92	34.44	41.45
UniXcoder	34.93	33.58	34.39	37.22
CodeGPT	35.37	34.11	32.15	37.61
CodeT5+	36.21	34.24	33.18	36.85
Contextor$_{LSTM}$	35.92	35.21	34.28	**42.67**
Contextor$_{Transformer}$	**36.96**	**36.97**	**36.31**	41.28

Table 6
Comparison of the results generated by two decoders under the same setting.

Model	BLEU	$BLEU_{weight}$	$Match_{ast}$	$Match_{df}$
Contextor$_{LSTM}$ L2R	35.92	35.21	34.28	42.67
Contextor$_{LSTM}$ R2L	35.79	35.01	34.32	42.31
Contextor$_{Transformer}$ L2R	36.96	36.97	36.31	41.28
Contextor$_{Transformer}$ R2L	35.88	35.12	33.21	41.59

results with fewer parameters. When we only use rewritten the dataset as the training data, we found that CodeT5+ and CodeGPT perform better. This is mainly because the decoders of these two models were pre-trained and learned more information. However, the decoders of other models are randomly initialized, and it is difficult to achieve good performances with a small amount of data. After using more mined data, Contextor achieved better results. We also observe that models generating ASTs achieve a higher score on CodeBLEU than BLEU, while CodeGPT and CodeT5+ score higher on BLEU. Compared with the code generated by pre-trained models, the code generated by seq2tree models gets 100% on syntactic validity, whereas the pre-trained model gets 92%. Thus, our results demonstrate the effectiveness of our proposed bidirectional decoding method and the superiority of seq2tree models for code generation tasks.

We also make a study on every part of CodeBLEU as shown in Table 5. Among models generating ASTs, Contextor can outperform other baselines in all the metrics because we employ a bidirectional decoder to better interact with both past and future knowledge. Although pre-trained models achieve relatively competitive results in BLEU and $BLEU_{weight}$, it has a disadvantage in $Match_{ast}$ because the code generated by these models cannot ensures the syntactic validity and code without syntactic validity score zero on $Match_{ast}$.

In Table 6, we compared the generation capabilities of two decoders. It can be observed that, when we applied the same criteria to both decoders, there is a small difference in their results, indicating that both decoders have the capability to generate high-quality code.

Summary for RQ1

Contextor can outperform the baselines in terms of two evaluation metrics on the CoNaLa dataset.

RQ2: How does bidirectional decoding mechanism facilitate the generation of context-coherent code? To demonstrate the effectiveness of the bidirectional decoding model, in this section, we will introduce how our bidirectional decoding mechanism facilitate the generation of context-coherent code.

Fig. 8 provide a concrete example, which illustrates how our bidirectional decoding model effectively generates context-consistent code. Specifically, given a natural language description *write dataframe df to a csv file*, our model is trained to generate the target code `df.to_csv(csvfile)`. At time step t_5, our L2R Decoder generates the method *to_csv* based on both the hidden state produced by the L2R Decoder and the context information generated by the R2L Decoder. Importantly, the other decoder has already generated its parameter *csvfile* when the L2R Decoder generates this method, providing additional information for the model.

We provide three attention heatmaps to prove our opinion. Fig. 9(a) describes the attention weights between the input sequence and the decoded results. When generating the node *to_csv*, more attention is attached to the words *csv* and *file* from the input. More importantly, Fig. 9(b) shows the attention weights between the decoders, demonstrating the relevance of past and future nodes. As shown in this figure, the $LSTM_{L2R}$ attaches more importance to the node *csvfile* as we anticipated. In addition, the heatmap of attention weights processed by the noise masking network is shown in Fig. 9(c). We can see that the noise masking network can mask out context information that is irrelevant to generation, thereby reducing the noise introduced by the bidirectional decoding mechanism.

To illustrate the superiority of the structure-level bidirectional decoding approach we proposed, we compared it with the traditional method of generating from both ends of the sequence, as shown in Table 7. Experimental results show that different traversal of the ASTs leads to improved generation results.

The decoding algorithm is crucial in NLG models, as it can expand the search space for model decoding. The Bidirectional Beam Search algorithm also has a significant impact in Contextor. As shown in Table 8, experimental results demonstrate that employing the bidirectional beam search algorithm leads to substantial performance improvements compared to the greedy algorithm.

Table 7
Comparison of Contextor using structure-level decoding methods with vanilla bidirectional decoding.

	BLEU	$BLEU_{weight}$	$Match_{ast}$	$Match_{df}$
Contextor$_{LSTM}$	35.92	35.21	34.28	42.67
Vanilla bidirectional LSTM	34.24	33.21	31.58	36.19
Contextor$_{Transformer}$	36.96	36.97	36.31	41.28
Vanilla bidirectional transformer	35.12	34.37	33.28	37.71

Table 8
Comparison between Bidirectional Beam Search algorithm and Greedy algorithm.

Model	Algorithm	BLEU	CodeBLEU
Contextor$_{LSTM}$	Greedy	30.67	32.19
	Bidirectional beam search	**35.92**	**37.02**
Contextor^{Transformer}	Greedy	31.00	30.46
	Bidirectional beam search	**36.96**	**37.88**

Table 9
Performance with different settings on CoNaLa.

Approach	BLEU	CodeBLEU
raw Contextor$_{LSTM}$	32.90	34.31
Contextor$_{LSTM}$ + Scheduled sampling	34.81	36.05
Contextor$_{LSTM}$ + Adapted scheduled sampling	**35.92**	**37.02**
raw Contextor$_{Transformer}$	33.72	35.91
Contextor$_{Transformer}$ + Scheduled sampling	35.58	36.23
Contextor$_{Transformer}$ + Adapted scheduled sampling	**36.96**	**37.88**

Summary for RQ2

Contextor can obtain valuable contextual information through bidirectional decoding.

RQ3: What is the role of adapted scheduled sampling?. To investigate the effectiveness of adapted scheduled sampling, we conducted an ablation study. We compared our original model with Contextor + scheduled sampling and Contextor + adapted scheduled sampling. The results are shown in Table 9.

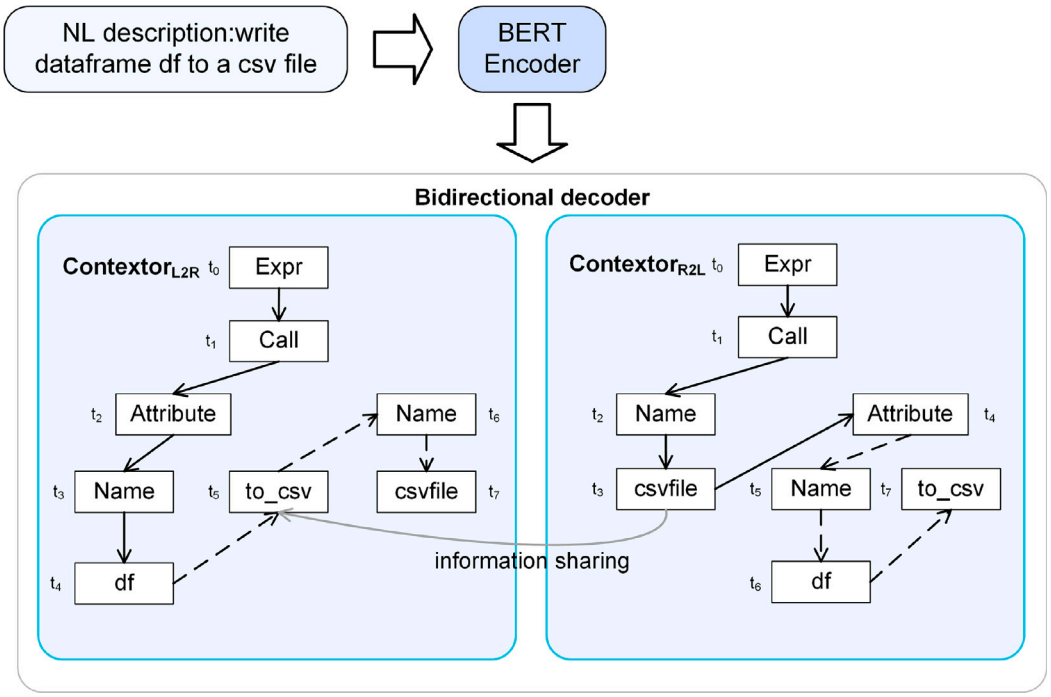
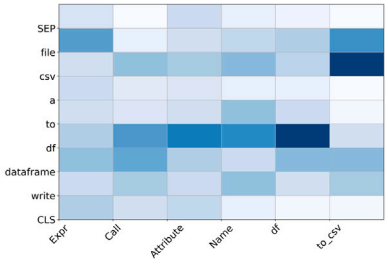
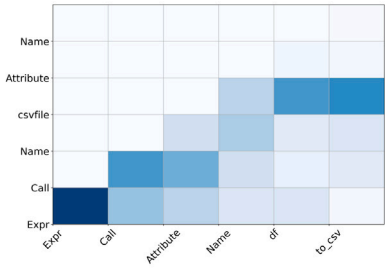


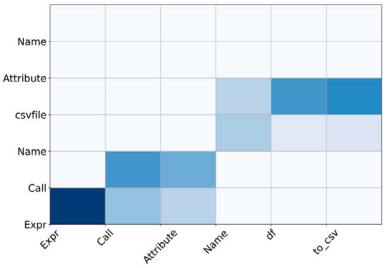
Fig. 8. An example of information sharing.



(a) Attention weights $\overrightarrow{weight}^{(enc)}$ between BERT encoder and LSTM$_{L2R}$



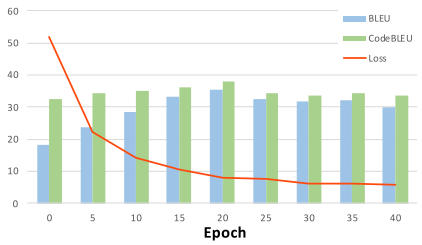
(b) Attention weights $\overrightarrow{weight}^{(dec)}$ between two decoders



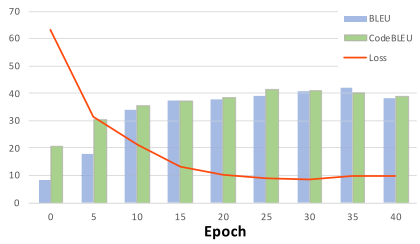
(c) Attention weights $\overrightarrow{weight}^{(dec)}$ between two decoders after noise masking

Fig. 9. Attention heatmaps.

The increase in BLEU and CodeBLEU can be explained in two ways. On the one hand, for sequence prediction tasks, the main difference between training and inference is whether use the actual previous token or the predicted token generated by the model. To be specific, conventional training approaches aim to minimize the loss given the current state and the actual previous token, while the token is replaced by the predicted token at inference which causes a discrepancy between training and inference. Scheduled sampling gently changes the training process from fully instructed by the actual target token to guided mainly by the predicted token generated by the model, narrowing the gap between training and inference. On the other hand, our bidirectional decoder suffer from the information leakage problem, as shown in Fig. 6. This problem is mainly caused by inputting the actual label into one decoder that is visible to the other. As a result, we employ scheduled sampling to gradually use the predicted label to train the model and alleviate information leakage. Considering that the tokens generated earlier are more likely to be leaked. So we continuously adapt scheduled sampling to use more predicted tokens at the earlier time step of generation. We also consider that the quality of the sequences generated in the later stage is lower than that in the initial



(a) Evaluation of the performance of Contextor$_{LSTM}$ w/o adapted scheduled sampling with different epochs on the CoNaLa.



(b) Evaluation of the performance of Contextor$_{LSTM}$ with different epochs on the CoNaLa.

Fig. 10. The impact of adapted scheduled sampling on the development set's BLEU and CodeBLEU scores, as well as the training set's loss.

Table 10
Evaluation of the decay parameters for Contextor$_{LSTM}$.

λ_1	λ_2	BLEU	CodeBLEU
0.98	0.1	33.91	34.81
	0.2	34.50	36.11
	0.3	35.12	35.91
0.95	0.1	34.68	36.21
	0.2	**35.92**	**37.02**
	0.3	34.31	36.26
0.92	0.1	35.03	36.29
	0.2	35.23	36.37
	0.3	34.79	35.03

Table 11
Evaluation of the decay parameters for Contextor$_{Transformer}$.

λ_1	λ_2	BLEU	CodeBLEU
0.98	0.1	34.16	33.92
	0.2	35.13	35.95
	0.3	36.08	37.22
0.95	0.1	35.97	35.31
	0.2	**36.96**	**37.88**
	0.3	35.19	36.13
0.92	0.1	35.29	35.81
	0.2	35.93	36.12
	0.3	34.85	36.24

stage (Zhou et al., 2019). Therefore, we use more actual tokens to guide our model in the later stage while generating a sequence. The ablation experiment shows that adapted scheduled sampling yields performance improvement.

Fig. 10 demonstrates the effect of adapted scheduled sampling on the bidirectional decoding mechanism. In Fig. 10(a), we observe that the model without adapted scheduled sampling achieves a quick convergence by encountering the true labels during training, but suffers from the issue of information leakage. As a result, its performance on BLEU and CodeBLEU on the validation set is relatively poor. On the other hand, Fig. 10(b) shows that the model using adapted scheduled sampling has a slower decrease in loss and even an upward trend in the later stages of training, but performs better on the validation set. This is because the use of adapted scheduled sampling mitigates the problem of information leakage, thereby reducing the gap between training and inference.

Decay parameters played a crucial role in our experiment as they determined the probability of selecting different guiding labels during sampling. As shown in the Table 10 and Table 11, Contextor$_{LSTM}$ and Contextor$_{Transformer}$ perform the best when using $\lambda_1 = 0.95$ and $\lambda_2 = 0.2$. This is because if λ is too large, the model relies too heavily on true labels, resulting in information leakage issues. On the other hand, if λ is too small, the model relies more on predicted labels, making it difficult to converge.

Summary for RQ3

We demonstrated the effectiveness of adapted scheduled sampling through an ablation study.

RQ4: What is the impact of different NL lengths, sequence lengths and code lengths on the performance of Contextor*?* To investigate how different NL lengths, sequence lengths and code lengths affect the performance of Contextor and the baselines, we analyzed the CodeBLEU metric scores of various action lengths generated by our proposed method and the baselines. Fig. 11 shows the average CodeBLEU scores of Contextor, BertranX, TRANX, and CodeT5+ for varying NL lengths, sequence lengths and code lengths.

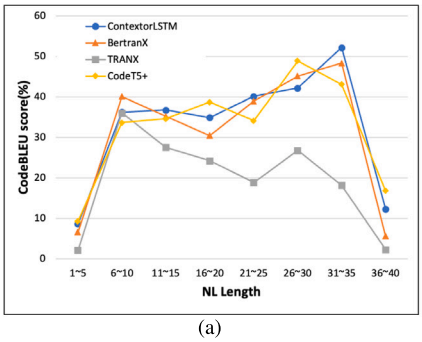
Fig. 11(a) illustrates that in cases where the NL input is lengthy, Contextor, BertranX, and CodeT5+ perform better owing to their pre-trained encoders. Additionally, BERT and CodeT5+, utilizing self-attention mechanisms, can capture long-range dependency information, making them better suited to processing longer inputs. From an overall perspective, as depicted in Figs. 11(b) and 11(c), BertranX performs better at generating shorter code and behavior sequences, while Contextor is superior at generating longer ones. This is due to the fact that a unidirectional decoding model generates the current character based on previously generated ones, causing errors to accumulate as the text lengthens. However, the bidirectional decoding model, Contextor, predicts not only based on the preceding context but also on the following context. In this situation, even if the previous prediction is erroneous, the model can rectify it based on the information supplied by the subsequent context, enhancing its code generation capabilities. Furthermore, compared to Contextor$_{LSTM}$, we observe that Contextor$_{Transformer}$ performs better in generating longer code. This is because the Transformer can capture dependencies over longer distances through its self-attention mechanism.

Summary for RQ4

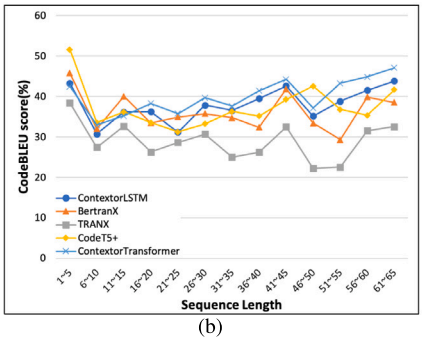
Contextor can generate higher quality code snippets compared to baseline methods in the case of relatively long input and target sequences.

4.2. Examples

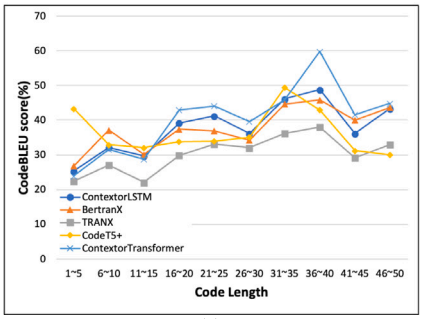
For qualitative analysis, we present examples of code generated by Contextor, BertranX, and CodeT5+ from Tables 12 to 14.



(a)



(b)



(c)

Fig. 11. CodeBLEU score for different NL lengths, sequence lengths and code lengths in CoNaLa dataset.

Table 12
Code snippets generated by Contextor, BertranX, and CodeT5+, where 'NL' is the input natural language, 'Gold' is the reference code snippet.

Example1	
NL	*Sort a list of dictionaries list_to_be_sorted by the value of the dictionary key 'name'*
Gold	newlist = sorted(list_to_be_sorted, key=lambda k:k['name'])
Contextor_{LSTM}	sorted(list_to_be_sorted, key=lambda x:x['name'])
$\text{Contextor}_{Transformer}$	sorted(list_to_be_sorted, key=lambda x:x['name'])
BertranX	sorted(list_to_be_sorted, key=lambda x:list(x.values()) [0])
CodeT5+	list = sorted(list_to_be_sorted, key=lambda x:x('name')

Table 13
Code snippets generated by Contextor, BertranX, and CodeT5+, where 'NL' is the input natural language, 'Gold' is the reference code snippet.

Example2	
NL	*Convert unix timestamp '1347517370' to formatted string '%Y-%m-%d %H:%M:%S'*
Gold	time.strftime('%Y-%m-%d%H:%M:%S', time.localtime(1347517370))
Contextor_{LSTM}	time.strftime('1347517370', time.strftime('1347517370', '%Y-%m-%d%H:%M:%S'))
$\text{Contextor}_{Transformer}$	time.strftime('%Y-%m-%d%H:%M:%S')
BertranX	time.mktime(time.mktime(datetime.datetime.fromtimestamp (s).timetuple()))
CodeT5+	time.strftime('%Y-%m-%d%H:%M:%S', time.localtime(137))

Table 14
Code snippets generated by Contextor, BertranX, and CodeT5+, where 'NL' is the input natural language, 'Gold' is the reference code snippet.

Example3	
NL	*append list 'list1' to 'list2'*
Gold	list2.append(list1)
Contextor_{LSTM}	list1.append(list1)
$\text{Contextor}_{Transformer}$	list2.append(list1)
BertranX	list2.append(list1)
CodeT5+	list2.append(list1)

In Table 12, it can be observed that the code generated by Contextor is very similar to the gold snippet, indicating that our approach can generate high-quality code corresponding to natural language. However, BertranX fails to interpret the natural language, and the output of CodeT5+ has a syntax problem. In Table 13, Contextor_{LSTM} has the problem of outputting duplicate code, which we attribute to the beam search algorithm (Su and Collier, 2022). BertranX is unable to output the corresponding code. Although CodeT5+ generates formally correct code, it did not successfully copy the parameters in natural language because of the out-of-vocabulary problem as we discussed in Section 3.1. As shown in Table 14, there is a logical problem with Contextor_{LSTM}'s output, while the other baselines generate the correct code. We think that the bidirectional decoding mechanism of Contextor occasionally introduces noise to the model because one decoder may output wrong or useless information that is irrelevant to the other decoder.

5. Threats to validity

Internal validity. Threats to internal validity are related to experimental errors and biases. To ensure a fair comparison with the baselines, we used a public dataset and followed the methodology of previous works (Yin and Neubig, 2018; Beau and Crabbé, 2022). We also reused their code and trained models (if available) to avoid possible errors.

Table 15
The comparison results between our proposed approach Contextor and baselines using scheduled sampling on CoNaLa dataset.

Approach	BLEU	CodeBLEU
TRANX	28.02	28.56
TRANX + scheduled sampling	29.03	29.31
BertranX	34.18	35.97
BertranX + scheduled sampling	34.43	35.82
Contextor_{LSTM}	35.92	37.02
$\text{Contextor}_{Transformer}$	36.96	37.88

External validity. Threats to external validity mainly focus on the generalizability of the results to other cases beyond the experimental settings. In our work, we selected the CoNaLa dataset as our experimental dataset, which consists of Python code mined from *Stack Overflow* and natural language descriptions mainly in the form of questions. We believe that this dataset represents a diverse set of coding scenarios that align with typical developer needs. Moreover, we note that generating code in CoNaLa is a challenging task due to the relatively low number of shared words between the natural language and code. Overall, we consider CoNaLa to be a representative corpus for evaluating code generation models.

Conclusion validity. As mentioned in RQ3, bidirectional decoding has its problem with information leakage, so we employ adapted scheduled sampling to alleviate this problem. However, models using scheduled sampling can also have a positive effect on the final result. As a result, to prove the effectiveness of the bidirectional decoding component of our work, we also employ scheduled sampling on TRANX and BertranX. Results in Table 15 can demonstrate the effectiveness of the bidirectional decoding mechanism.

Construct validity. Finally, we discuss threats to construct validity, which refers to the appropriateness of the automatic evaluation metrics. Previous works have commonly used BLEU as their evaluation metric for code generation tasks. However, as we discussed in RQ4, similarity between the predicted code and the target code cannot fully represent the quality of the code. Thus, we opted to use CodeBLEU to evaluate the results, which considers not only surface matching but also grammatical and logical correctness.

6. Related work

Code generation. Automatic generation of code from natural language has received increasing attention in software engineering and artificial intelligence.

Ling et al. (2016) used Latent Predictor Networks to generate general-purpose code from a mixed natural language and structured specification, which treat code generation as a sequence-to-sequence problem. Rabinovich et al. (2017) proposed abstract syntax networks for code generation and semantic parsing, which utilized a modular decoder to generate ASTs in top-down order. They put forward a sequence-to-tree framework and combined the decoder with the attention mechanism to integrate the information of input natural language. Yin and Neubig (2018) proposed TRANX, employing a transition system to generate domain-dependent grammar and construct such grammar to code. This method ensures syntactic correctness, and their transition system can be extended to other domain-specific languages. Beau and Crabbé (2022) replaced the LSTM encoder in TRANX with a BERT encoder, which improves the ability to understand the input natural language. They also made a detailed ablation study to demonstrate the impact of each component. Sun et al. (2020) proposed TreeGen, which uses self-attention mechanism of Transformers (Vaswani et al., 2017b) to alleviate the long-dependency problem. Based on such a sequence-to-tree model, Hayati et al. (2018) proposed RECODE, a retrieval-based approach that extracts subtrees from retrieved code. Experiments show that RECODE can benefit from the information from the retrieved code. Jiang et al. (2021a) proposed a novel AST structure enhanced decoder for code generation, which model the predictions of current actions and future actions via multi-task learning. Wei et al. (2019) applied the relation between code generation (CG) and code summarization (CS) via dual learning. They designed a dual learning framework to train CG and CS frameworks simultaneously to boost each other. Yang et al. (2022) propose a novel template-augmented exploit code generation approach Exploit-Gen based on CodeBERT. They used two encoders for encoding and a fusion layer to fuse the information of the two encoders, which improve the quality of the generation.

Besides, with the effectiveness and popularity of the pre-trained models, researchers began to pre-train NL and code on Transformer models in various ways. CodeBERT, proposed by Norouzi et al. (2021), is pre-trained on the Code Search Net dataset (Husain et al., 2019). This task consists of 6 million code functions and 2 million function-documentation pairs collected from open-source projects. This corpus contains about 6 million functions from open-source code among six programming languages and 2 million function-documentation pairs collected from open-source projects. They used masked language modeling (MLM) and replaced token detection (RTD) to train the model. Lu et al. (2021a) proposed CodeGPT. CodeGPT is based on the architecture of GPT-2 (Radford et al., 2019), which is a decoder-only model. CodeT5 (Wang et al., 2021) is an encoder–decoder model pre-trained on Transformers (Vaswani et al., 2017b) that take token type prediction as a pre-training task. CodeT5 is based on T5 architecture that has been demonstrated to improve code generation tasks.

Bidirectional decoding. Most NLG tasks generate a sequence in an autoregressive way, such as neural machine translation, code generation, and code summarization which can be formalized as a sequence-to-sequence problem. Generally, most previous works generate the target sequence in a right-to-left manner. However, such a manner can only utilize the previous knowledge to predict the next token that ignores the future.

Watanabe and Sumita (2002) first explored bidirectional decoding by decoding sequence in both left-to-right and right-to-left manner independently and merging hypothesized partial outputs of two directions. Su et al. (2019) leveraged the reverse target-side contexts to enhance neural machine translation through asynchronous bidirectional decoding. Zhang et al. (2018) proposed ABD-NMT to fully exploit the source-side and target-side contexts to improve generation quality. They equipped the conventional sequence-to-sequence model with a backward decoder that independently generates the target sequence in a right-to-left manner. Then they generate the sequence from left to right, which employs the attention mechanism using the hidden state from both the encoder and backward decoder. Zhang et al. (2020) proposed a synchronous bidirectional decoding model that generates sequences using both left-to-right and right-to-left decoding results simultaneously and interactively. Unlike Zhang's (Zhang et al., 2018) work, their model generates left-to-right and right-to-left sequences simultaneously, and both decoders generate sequences instructed by the encoder's output and the other's hidden states.

7. Conclusion

In this paper, we have presented Contextor, a sequence-to-tree model with a context-sensitive bidirectional decoder for code generation. Different from previous work, we use the bidirectional decoding mechanism to support context-awareness at the code structure level. Experiment results have shown that our approach outperforms the state-of-the-art baselines on the public dataset. In addition, we use the scheduled sampling technique to mitigate the information leakage problem of the bidirectional decoding mechanism, which we believe is of interests in its own right.

In the future, we plan to deploy our method on different model architectures, such as Transformer (Vaswani et al., 2017a) and RWKV (Peng et al., 2023), and attempt to further improve the model's performance by increasing the number of parameters.

CRediT authorship contribution statement

Xiangyu Zhang: Writing – review & editing, Writing – original draft, Software, Methodology, Data curation, Conceptualization. **Yu Zhou:** Writing – review & editing, Supervision, Funding acquisition, Formal analysis. **Guang Yang:** Methodology, Investigation, Data curation. **Tingting Han:** Writing – review & editing, Formal analysis. **Taolue Chen:** Writing – review & editing, Funding acquisition, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62372232), the Fundamental Research Funds for the Central Universities, China (No. NG2023005), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, China. T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University, China (KFKT2022A03), Birkbeck BEI School Project (EFFECT), China.

References

Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D., 2016. Deepcoder: Learning to write programs. arXiv preprint arXiv:1611.01989.

Beau, N., Crabbé, B., 2022. The impact of lexical and grammatical processing on generating code from natural language. In: Muresan, S., Nakov, P., Villavicencio, A. (Eds.), Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022. Association for Computational Linguistics, pp. 2204–2214.

Bengio, S., Vinyals, O., Jaitly, N., Shazeer, N., 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. pp. 1171–1179.

Devlin, J., Chang, M., Lee, K., Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (Eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, pp. 4171–4186.

Earley, J., 1970. An efficient context-free parsing algorithm. Commun. ACM 13 (2), 94–102.

Gu, J., Bradbury, J., Xiong, C., Li, V.O., Socher, R., 2017. Non-autoregressive neural machine translation. arXiv preprint arXiv:1711.02281.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., GraphCodeBERT: Pre-training code representations with data flow. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.

Hayati, S.A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., Neubig, G., 2018. Retrieval-based neural code generation. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018. Association for Computational Linguistics, pp. 925–930.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Hussain, Y., Huang, Z., Zhou, Y., Wang, S., 2020. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. Inf. Softw. Technol. 125, 106309. http://dx.doi.org/10.1016/j.infsof.2020.106309.

Jiang, H., Song, L., Ge, Y., Meng, F., Yao, J., Su, J., 2021a. An AST structure enhanced decoder for code generation. IEEE/ACM Trans. Audio, Speech, Lang. Process. 30, 468–476.

Jiang, H., Zhou, C., Meng, F., Zhang, B., Zhou, J., Huang, D., Wu, Q., Su, J., 2021b. Exploring dynamic selection of branch expansion orders for code generation. arXiv preprint arXiv:2106.00261.

Kou, B., Chen, S., Wang, Z., Ma, L., Zhang, T., 2023. Is model attention aligned with human attention? An empirical study on large language models for code generation. arXiv preprint arXiv:2306.01220.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K.M., Kociský, T., Wang, F., Senior, A.W., 2016. Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S., 2021a. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In: Vanschoren, J., Yeung, S. (Eds.), Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, Virtual.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021b. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

Norouzi, S., Tang, K., Cao, Y., 2021. Code generation from natural language with less prior knowledge and more monolingual data. In: Zong, C., Xia, F., Li, W., Navigli, R. (Eds.), Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021. Association for Computational Linguistics, pp. 776–785.

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

Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Cao, H., Cheng, X., Chung, M., Grella, M., GV, K.K., et al., 2023. RWKV: Reinventing RNNs for the transformer era. arXiv preprint arXiv:2305.13048.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, pp. 1139–1149.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S., 2020. CodeBLEU: a method for automatic evaluation of code synthesis. CoRR abs/2009.10297, arXiv:2009.10297.

Stahlberg, F., 2020. Neural machine translation: A review. J. Artificial Intelligence Res. 69, 343–418.

Su, Y., Collier, N., 2022. Contrastive search is what you need for neural text generation. arXiv preprint arXiv:2210.14140.

Su, J., Zhang, X., Lin, Q., Qin, Y., Yao, J., Liu, Y., 2019. Exploiting reverse target-side contexts for neural machine translation via asynchronous bidirectional decoding. Artificial Intelligence 277, 103168.

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019. A grammar-based structural CNN decoder for code generation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, pp. 7055–7062. http://dx.doi.org/10.1609/aaai.v33i01.33017055.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L., 2020. TreeGen: A tree-based transformer architecture for code generation. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, pp. 8984–8991.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017a. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017b. Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA. pp. 5998–6008.

Wang, D.C., Appel, A.W., Korn, J.L., Serra, C.S., 1997. The zephyr abstract syntax description language. In: Ramming, C. (Ed.), Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997. USENIX, pp. 213–228.

Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C., 2023. Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922.

Wang, Y., Wang, W., Joty, S.R., Hoi, S.C.H., 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021. Association for Computational Linguistics, pp. 8696–8708.

Wang, W., Zhang, K., Li, G., Liu, S., Jin, Z., Liu, Y., 2022. A tree-structured transformer for program representation learning. arXiv preprint arXiv:2208.08643.

Watanabe, T., Sumita, E., 2002. Bidirectional decoding for statistical machine translation. In: 19th International Conference on Computational Linguistics, COLING 2002, Howard International House and Academia Sinica, Taipei, Taiwan, August 24 - September 1, 2002.

Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z., 2019. Code generation as a dual task of code summarization. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 6559–6569.

Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al., 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144.

Xie, B., Su, J., Ge, Y., Li, X., Cui, J., Yao, J., Wang, B., 2021. Improving tree-structured decoder training for code generation via mutual learning. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. AAAI Press, pp. 14121–14128.

Yang, Z., Chen, S., Gao, C., Li, Z., Li, G., Lv, R., 2023a. Deep learning based code generation methods: A literature review. CoRR abs/2303.01056.

Yang, G., Zhou, Y., Chen, X., Yu, C., 2021. Fine-grained pseudo-code generation method via code feature extraction and transformer. In: 2021 28th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 213–222.

Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2022. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 111577. http://dx.doi.org/10.1016/j.jss.2022.111577.

Yang, G., Zhou, Y., Chen, X., Zhang, X., Xu, Y., Han, T., Chen, T., 2023b. A syntax-guided multi-task learning approach for turducken-style code generation. arXiv preprint arXiv:2303.05061.

Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G., 2018. Learning to mine parallel natural language/source code corpora from stack overflow. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (Eds.), Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp. 388–389.

Yin, P., Neubig, G., 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Blanco, E., Lu, W. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018. Association for Computational Linguistics, pp. 7–12.

Zhang, A., Fang, L., Ge, C., Li, P., Liu, Z., 2023. Efficient transformer with code token learner for code clone detection. J. Syst. Softw. 197, 111557. http://dx.doi.org/10.1016/j.jss.2022.111557.

Zhang, X., Su, J., Qin, Y., Liu, Y., Ji, R., Wang, H., 2018. Asynchronous bidirectional decoding for neural machine translation. In: McIlraith, S.A., Weinberger, K.Q. (Eds.), Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. AAAI Press, pp. 5698–5705.

Zhang, J., Zhou, L., Zhao, Y., Zong, C., 2020. Synchronous bidirectional inference for neural sequence generation. Artificial Intelligence 281, 103234.

Zhou, L., Zhang, J., Zong, C., 2019. Synchronous bidirectional neural machine translation. Trans. Assoc. Comput. Linguist. 7, 91–105.

Xiangyu Zhang is currently pursuing the master degree with the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation.

Yu Zhou is currently a full professor of software engineering in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his Ph.D. in computer science from Nanjing University in 2009 supervised by Professor Jian Lü. From 2006–2007, he worked as a research assistant with Professor Jiannong Cao in Department of Computing at Hongkong Polytechnic University. From 2007–2008, he was funded by a joint Ph.D. education program from China Scholarship Council and studied at University of Zurich, supervised by Professor Harald Gall. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politecnico di Milano, Italy, working with Professor Luciano Baresi. From 2015–2016, he visited SEAL lab at University of Zurich, where he is also an adjunct researcher. He is currently a senior member of IEEE/CCF, a member of Technical Committee (TC) on System Software of CCF, a member of TC on Software Engineering of CCF, and vice director of TC on Software of Jiangsu Computer Society. He has broad interests in software engineering with a focus on intelligent software engineering, big data and cloud computing, software evolution and reliability analysis, and co-authored more than 100 papers in these fields.

Guang Yang is currently pursuing the Ph.D. degree with the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation and exploit code.

Tingting Han obtained her B.Sc. and MEng in Computer Science from Nanjing University China, and her Ph.D. from RWTH Aachen University and University of Twente. She was a Research Assistant at University of Oxford before joining Birkbeck. Her Areas of interest: Formal verification and synthesis of probabilistic systems, and its applications.

Taolue Chen is a Postdoctoral Researcher at University of Oxford (UK) and University of Twente (The Netherlands);Ph.D. (CWI and Vrije Universiteit Amsterdam, The Netherlands), Master and Bachelor (Nanjing University, China), all in Computer Science. His Areas of interest: Quantitative analysis and synthesis of computer program and systems, logic in computer science, machine learning and data science, software engineering, algorithms and computational complexity.