Abstract—Adapting deep learning (DL) techniques to automate nontrivial coding activities, such as code documentation and defect detection, has been intensively studied recently. Learning to predict code changes is one of the popular and essential investigations. Prior studies have shown that DL techniques, such as neural machine translation (NMT), can benefit meaningful code changes, including bug fixing and code refactoring. However, NMT models may encounter bottleneck when modeling long sequences; thus, they are limited in accurately predicting code changes. In this article, we design a Transformer-based approach, considering that the Transformer has proven effective in capturing long-term dependencies. Specifically, we propose a novel model named DTrans. For better incorporating the local structure of code, i.e., statement-level information, DTrans is designed with dynamically relative position encoding in the multhead attention of the Transformer. Experiments on benchmark datasets demonstrate that DTrans can more accurately generate patches than the state-of-the-art methods, increasing the performance by at least 5.45–46.57% in terms of the exact match metric on different datasets. Moreover, DTrans can locate the lines to change with 1.75–24.21% higher accuracy than the existing methods.

Index Terms—Code edit, position encoding, Transformer.

I. INTRODUCTION

D
ing deep learning (DL) techniques have been adapted to solve many traditional software engineering problems and tasks recently [1], [2], e.g., fault localization [3], [4], [5], automatic program repair [6], [7], [8], code summarization [9], [10], [11], code prediction [12], [13], and defect prediction [14], [15], [16].

Dynamically Relative Position Encoding-Based Transformer for Automatic Code Edit

Shiyi Qi, Yaoxian Li, Cuiyun Gao, Xiaohong Su, Shuzheng Gao, Zibin Zheng, and Chuanyi Liu

Manuscript received 29 November 2021; revised 23 April 2022; accepted 8 July 2022. Date of publication 22 August 2022; date of current version 1 September 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 620202084, Grant 61872110, and Grant 61672191, in part by the Stable Support Plan for Colleges and Universities in Shenzhen under Grant GXWD2020 1230155427003-20200730108139009, in part by the Major Key Project of Peng Cheng Laboratory under Grant PCL2022A03, Grant PCL2021A02, and Grant PCL2021A09, in part by the Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies under Grant 2022B1212010005, and in part by the Science and Technology Program of Guangzhou, China, under Grant 202103050004. Associate Editor: He Jiang. (Corresponding authors: Cuiyun Gao; Chuanyi Liu.)

Shiyi Qi, Yaoxian Li, Xiaohong Su, and Shuzheng Gao are with the Harbin Institute of Technology, Harbin 150001, China (e-mail: sqyi081125@163.com; yaoxian0803@icloud.com; ssx@hit.edu.cn; szgao98@gmail.com).

Cuiyun Gao and Chuanyi Liu are with the Harbin Institute of Technology, Harbin 150001, China, and also with the Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen 518066, China (e-mail: gaocuiyun@hit.edu.cn; liuchuanyi@hit.edu.cn).

Zibin Zheng is with Sun Yat-sen University, Guangzhou 510275, China (e-mail: zibinzheng2@yeah.net).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3194370.

Digital Object Identifier 10.1109/TR.2022.3194370

Among these fields, learning from code for code change prediction draws more and more research investigations [17], [18]. Precisely editing code can significantly facilitate the software maintenance process for developers [19], [20].

In the process of program development and maintenance, developers usually need to modify the source code for various reasons, including program repair [21], [22], code refactoring [23], [24], [25], and application programming interface (API)-related changes [26], [27]. Such behavior is known as “code edit” or “code change” [19], [20]. Prior research [19], [20], [28], [29] discovers that code edits generally follow repetitive edit patterns and can be employed to automatically generate the targeted code based on the original code. Fig. 1 shows two examples for illustrating the code edit task. In the original code of Fig. 1(a), the method testEmpty needs to return the object’s ID and name. However, the functions getId() and getName() do not exist for the object, which leads to a program bug. The correct code edit operation is to generate a correct patch for fixing the bug, i.e., changing to the corresponding correct functions getId() and getName(), respectively. For the example in Fig. 1(b), the parameter name is changed from type to method for enhancing the readability of the code. The code edit task aims at generating the edited code given the original code [19]. Due to the complex code edit patterns, automatically identifying the lines for editing and producing accurate edits are challenging.

In recent years, DL has made great progress and been applied to many code-related tasks [30], [31]. The large software engineering datasets, such as GitHub that includes over 100 million repositories with over 200 million merged pull requests (PRs) and 2 billion commits [19], provide us with sufficient source code for training DL models. Prior studies [19], [29], [32] have shown that DL techniques, such as neural machine translation (NMT) [33], can automatically apply developers’ PR code to generate meaningful code changes. NMT models treat PR code as a series of tokens or use the parsed tree structure as input, then creating an intermediate representation with an encoder network and decoding the representation into target sequence with a decoder network [9], [34]. This mechanism makes NMT models learn complex code change patterns between input and output sequences [20]. However, NMT models have proven ineffective in modeling long sequences [35]; thus, they may be limited in accurately editing code. Considering that the Transformer [36], [37], [38] has shown more effective than NMT in modeling long sequences, it is more applicable for the task. But directly adopting the Transformer still cannot well capture the structural dependencies between tokens [9], [39].
Thus, to mitigate the issue of NMT models and better capture the code dependencies, we propose a novel Transformer-based model, named as DTrans. Prior research [20] extracts the abstract syntax tree (AST) of the original code for capturing the structural information. In this work, to alleviate the complexity caused by the AST extraction, we focus on exploiting the local structure, i.e., the statement-level information, which can be easily obtained without parsing. Besides, for the code editing task, the changes generally happen within several statements, indicating the importance of local structure [20], [29]. Specifically, we propose a dynamically relative position encoding strategy to integrate the variational statement-level information. Different from the Transformer [36] and the Transformer with relative position [40], which represent position embedding by absolute position and relative position, respectively, DTrans conducts positional encoding guided by statement information.

To evaluate the performance of our proposed DTrans model, we choose three PR repositories utilized by the work [19] as benchmark datasets, including Android [41], Google Source [42], and Ovirt [43]. Besides, we also involve the 2.3 million 121 895 pairwise code changes from GitHub open-source projects [32]. During evaluation, we group project datasets to two levels, i.e., small and medium levels, according to the token numbers of the original code following prior studies [29], [32], [44]. Experiments demonstrate that DTrans accurately predicts more code edits in both small-level and medium-level projects, increasing the performance of the best baseline [29] by at least 5.45% and 25.76% in terms of the exact match metric, respectively. Moreover, DTrans successfully locates the lines to change with 1.75–24.21% higher accuracy than the best baseline.

Overall, we make the following contributions.

1) A novel Transformer-based approach is proposed to incorporate a dynamically relative position encoding strategy in the multihead attention of the Transformer, which explicitly incorporates the statement-level syntactic information for better capturing the local structure of the code.

2) We evaluate our approach on benchmark datasets, and the results demonstrate the effectiveness of DTrans in predicting accurate code changes.

**Article structure:** We introduce the background in Section II. The proposed approach is illustrated in Section III. The experimental setup and results are depicted in Sections IV and V, respectively. We show some cases in Section VI. The threats to validity and related work are introduced in Sections VII and VIII, respectively. Finally, Section IX concludes this article.

II. BACKGROUND

In this section, we first formulate the code change prediction task and then introduce the basic approach—Transformer.

A. DL in Code Change Prediction

DL-based techniques aim at learning the mapping relations between the original code and the target code by training and generating the edited code for facilitating software development [29], [45], [46]. Programming languages can be treated as sequences of code tokens. Therefore, the problem of code change prediction can be tackled as an NMT problem [20], [32], that is, to “translate” from a sequence of code tokens (the original code) to another sequence of code tokens (the target code).

We take a sequence of the original code \( O \) as an example, and let

\[
O = (o_1, o_2, \ldots, o_i, \ldots o_m)
\]

where each \( o_i \) is the \( i \)th token in the code. Each input sequence \( O \) corresponds to a target code \( C \), denoted as

\[
C = (c_1, c_2, \ldots, c_n)
\]

where \( m \) and \( n \) indicate the lengths of the original and target sequences, respectively. Our goal is to learn the conditional distribution and generate changed code sequence by maximizing
the conditional likelihood
\[ C = \arg \max_C P(C|O). \]
Finally, we achieve an optimized target sequence as the predicted code change.

B. Transformer

The Transformer employs the typical encoder–decoder structure [44] and is composed of stacked Transformer blocks. Each block contains a multihead self-attention sublayer followed by a fully connected positionalwise feedforward network (FFN) sublayer. The sublayers are connected by residual connections [47] and layer normalization [48]. In addition, the Transformer augments the input features by adding a positional embedding since the self-attention mechanism lacks a natural way to encode the word position information. The Transformer also applies pad masking to resolve the problem of variable input lengths, and its decoder uses sequence masking in its self-attention to ensure that the predictions for the ith position can only depend on the known outputs at positions less than i. We introduce the major components of the Transformer, including multihead self-attention, positionwise FFNs, and basic blocks of the Transformer in the following.

1) Multihead Self-Attention: Multihead self-attention involves multiple attention heads and performs self-attention mechanism on every head. One attention head obtains one representation space for the same text, and multihead attention obtains multiple different representation spaces. The self-attention mechanism can be described as mapping a query and a set of key–value pairs to an output, where the query, key, value, and output are all d-dimensional vectors. The output of each head is concatenated, and results in the final output vector once again projected.

   a) Scaled dot-product attention: The self-attention used in the Transformer is also known as scaled dot-product attention. Scaled dot-product attention aims to pay more attention to the important information of input sequence [36]. It transposes the sequence of input vectors \( \mathbf{X} = (x_1, x_2, \ldots, x_n) \) into the sequence of output vectors \( \mathbf{Z} = (z_1, z_2, \ldots, z_n) \), where \( x_i, z_i \in \mathbb{R}^{d_{\text{head}}} \). When doing self-attention, the Transformer first projects the input vector \( \mathbf{X} \) into three vectors: the query \( \mathbf{Q} \), key \( \mathbf{K} \), and value \( \mathbf{V} \) by trainable parameters \( \mathbf{W}_Q, \mathbf{W}_K, \) and \( \mathbf{W}_V \). The attention weight is calculated using the dot product and the softmax function. The output vector is the weighted sum of the value vector

   \[
   e_{ij} = \frac{(x_i \mathbf{W}_Q^T)(x_j \mathbf{W}_K)^T}{\sqrt{d}} \quad (1)
   \]

   \[
   \alpha_{ij} = \frac{\exp (e_{ij})}{\sum_{k=1}^n \exp (e_{ij})} \quad (2)
   \]

   \[
   z_i = \sum_{j=1}^n \alpha_{ij}(x_j \mathbf{W}_V) \quad (3)
   \]

   where \( d \) is the dimension of each vector and is used to scale the dot product.

   b) Multihead attention: Multihead attention captures different contexts with multiple individual self-attention functions. This mechanism allows the Transformer to jointly attend to information from different representation subspaces. Multihead attention is computed after scaled dot-product attention

   \[
   \text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \mathbf{W}_O \quad (4)
   \]

   \[
   \text{head}_i = \text{Attention}(\mathbf{XW}_i^Q, \mathbf{XW}_i^K, \mathbf{XW}_i^V) \quad (5)
   \]

   where \( \mathbf{W}_O \) indicates the learnable parameters and the parameters \( \mathbf{W}_i^Q, \mathbf{W}_i^K, \) and \( \mathbf{W}_i^V \) are independent in each head.

2) Positionwise FFNs: In addition to multihead self-attention sublayers, each block in the encoder and the decoder also contains a fully connected FFN sublayer. The FFN transforms the current feature space into another space through nonlinear mapping, aiming at learning a better representation of the input. The parameters of each position are shared. This FFN can be computed by two linear transformations and a rectified linear unit activation function between them

   \[
   \text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \quad (6)
   \]

   where \( W_1, W_2, b_1, \) and \( b_2 \) are learnable parameters.

3) Basic Blocks of the Transformer: The Transformer is composed of stacked encoder–decoder blocks. Every block in the Transformer has a multihead self-attention sublayer and an FFN sublayer. These sublayers are connected by the residual connections (see [47]) and layer normalization (see [48]). Different from the encoder block, the decoder block has other attention sublayers that use the key and value matrices from the encoder instead of calculating them from the projection of input (DTrans is a Transformer-based architecture, so the structure of the encoder and decoder block can also refer to Fig. 2). Besides, the number of encoder–decoder blocks will affect the performance of the Transformer, and more encoder–decoder blocks will increase the model size and require more time to train.

III. METHODOLOGY

In this section, we introduce the Transformer-based model DTrans for automatic code change prediction. The overall architecture of DTrans is shown in Fig. 2, following the general Transformer framework (as introduced in Section II). In order to mitigate the out-of-vocabulary (OOV) problem, we first perform code abstraction following the prior work [32], [49]. Also, different from the vanilla Transformer, we propose a novel position encoding strategy, named dynamically relative position encoding, to incorporate statement-level syntactic information into the Transformer for better capturing the local structure of code. We elaborate on the code abstraction process and the proposed dynamically relative position encoding in more details in the following.

A. Code Abstraction

Different from natural language, tokens in programming language are more diverse since developers can define variable names and function names in variant ways. The diversity
of identifiers and literals in the code leads to a more serious OOV problem during program comprehension. Thus, following Ahmed et al. [49] and Tufano et al.’s [32] good practice, we adopt code abstraction to reduce vocabulary size and mitigate the OOV problem.

An example of code abstraction is shown in Fig. 3. Specifically, we use src2abs provided in [19] and [32] to abstract the source code. It feeds the sequence of the source code to a Java parser [50], which can recognize identifiers and literals and then generate and substitute a unique ID for each identifier and literal. If the identifier or literal appears multiple times in the same source code, it will be replaced with the same ID. Since some identifiers and literals appear frequently in the source code, they can be treated as keywords of the dataset [32]. The frequent identifiers and literals should not be abstracted but regarded as idioms that src2abs has provided for us.

B. Dynamically Relative Position Representations

1) Relation-Aware Self-Attention: Using different position embeddings for different positions helps the Transformer capture the position information of input words. However, absolute positional encoding in the vanilla Transformer is ineffective to capture the relative word orders [40]. To encode the pairwise positional relationships between input elements, Shaw et al. [40] propose the relative position encoding, which models the relation of two elements through their distance in the input sequence. Formally, the relative position embedding between input element $x_i$ and $x_j$ is represented as $a_{ij}^V, a_{ij}^K \in \mathbb{R}^d$.

In this way, the self-attention calculated in (1) and (3) can be rewritten as

$$
e_{ij} = \frac{\langle x_i W^Q \rangle \langle x_j W^K + a_{ij}^K \rangle^T}{\sqrt{d_z}}$$

$$z_i = \sum_{j=1}^n \alpha_{ij} \left( x_j W^V + a_{ij}^V \right).$$

Shaw et al. [40] also clip the maximum relative position to a maximum absolute value of $k$ since they hypothesize that precise relative position information is not useful beyond a certain
is in the same statement, the value 
+ 1 receives the two kinds of attention, while the last 
= ( ) 
= 3 relative position representations,
= 0 
= (12) 
= 0 max 
= a 
as example for illustrating the 
[denoted as the 
in 
s( 
is bigger than 
compared 
is an 
′ ∈ 
L 
n 
R 
(10) 
between them 
= 
=, 
= 1, 
= 1 
and 
= ( 
is defined as 3. For the token 
(9) 
is set as 0. We compute the 
= 
and 
= a 
V where 
)) 
K 2
Hence, we learn 2k + 1 relative position representations, 
i.e., 
K 
= (w K k ,..., w K k ) and 
V 
= (w V k ,..., w V k ) where 
K , 
V ∈ R d model.

2) Dynamically Relation Position: 
The relative position en-
coding [40] captures the pairwise positional relationships 
between the input elements by simply fixing the maximum relative 
distance at k. To involve the local structure information of code, 
we propose to incorporate the statement-level information into 
the position encoding. Different from predefining a maximum 
clipping distance k, we propose to dynamically adjust the dis-
tance based on the length of the statement during the relative 
position encoding, named as dynamically relation position en-
coding. The difference between relative position embedding and 
the proposed strategy is illustrated in Fig. 4, and the clipping 
distance k is defined as 3. For the token 
VAR_1, the relative 
position encoding enhances the relationship among the tokens 
before and behind the token 
VAR_1, which is indicated with 
dotted lines in the relative position encoding method of Fig. 4(a).

We hypothesize that tokens in one statement have stronger rela-
tions with the tokens in other statements, e.g., the token 
VAR_1 tends to be weakly relevant to the token 
METHOD_1 compared 
with the tokens in the same statement (e.g., token 
METHOD_2).

To incorporate statement-level syntactic information into the 
position embedding, we propose a dynamically relation position 
encoding strategy. The proposed position encoding can help 
the Transformer pay more attention to the tokens in the same 
statement [denoted as the solid lines in Fig. 4(b)] and the
tokens with a relative distance smaller than k [denoted as the 
dotted lines in Fig. 4(b)]. In addition, the two kinds of attention 
can be superimposed in our strategy. For example, the token 
METHOD_2 receives the two kinds of attention, while the last 
two tokens ”) ” and ”; ” do not receive the relative position 
attention because their relative distance to 
VAR_1 is bigger than 
the clipping distance k (k = 3 here). In the decoder, the current 
token cannot see the token behind, so it is impossible to get the 
statement mask matrix. Therefore, we only use the dynamically 
relative position in the encoder.

Similar to padding mask and sequence mask, we propose a 
statement mask operation to divide the code into a sequence of 
statements. For the code example shown in Fig. 5, we illustrate 
the statement mask matrix for its statements 
S1, S2, and 
S3 in 
Fig. 2. Specifically, the statement mask matrix 
W L is an n × n 
matrix, which records the statement-aware information of the 
source code, where 
n is the length of code tokens. For the tokens 
i, and 
j in the same statement, the value 
W L ij between them is set as 1; otherwise, the value 
W V L ij is set as 0. We compute the 
dynamically relative position embeddings as follows:

\[
\begin{align*}
  z_t &= \left\{ \begin{array}{ll}
  \sum_{j=1}^{n} a_{ij} (x_j W^V + a_{ij}^V) , & W^L_{ij} = 1 \\
  \sum_{j=1}^{n} a_{ij} (x_j W^V + a_{ij}^V) , & W^L_{ij} = 0
\end{array} \right. 
\end{align*}
\]

Fig. 4. Illustration of the token relations for (a) relative positions and (b) dynamically relative positions. For each relative position, the clipping distance k is assumed as 3. Only the second and third statements of the source code are illustrated here for simplicity. Solid lines and dotted lines indicate different token relations. The dotted line represents the relative distance is smaller than k, and the solid line represents the tokens in the same statement with 
VAR_1.

Fig. 5. Example of the source code to represent statement mask matrix. Note that we only take statements 
S1, 
S2, and 
S3 as example for illustrating the statement mask matrix in Fig. 2.
where $W' \in R^{n \times n}$ is the statement mask matrix, and $a^V \in R^{4_{\text{model}}}$ is a learnable parameter vector. We then recalculate the attention weight to incorporate the dynamically relative position embeddings

$$e_{ij} = \begin{cases} x_i W^Q(x_j W^K + a^H_k + a^W_{ij})^T, & W_{ij}^L = 1 \\ x_i W^Q(x_j W^K + a^H_k + a^W_{ij})^T, & W_{ij}^L = 0 \end{cases}$$  \tag{13}$$

where $a^W \in R^{4_{\text{model}}}$ is a learnable parameter vector.

IV. EXPERIMENTAL SETUP

In this section, we will introduce the benchmark datasets, implementation details, evaluation metrics, and comparison models for experimentation.

A. Benchmark Datasets

We conduct evaluation on two benchmark datasets: following the previous work [19], [28], Gerrit code reviews repository, and open-source projects in GitHub (namely GitProjs). Gerrit includes Android [41], Google Source [42], and Ovirt [43], while GitProjs contains 121 895 PRs commits from GitHub open-source projects. We classify all the projects in the datasets into two levels, i.e., small level $M_{\text{small}}$ and medium level $M_{\text{medium}},$ according to the tokens numbers of original code. $M_{\text{small}}$ and $M_{\text{medium}}$ contain 0–50 tokens and 500–100 tokens in each piece of original code, respectively. The two benchmark datasets are partitioned into training set (80%), validation set (10%), and test set (10%) following the prior studies, with detailed statistics shown in Table I.

B. Implementation and Supporting Tools/Platform

1) Data Preparation: We first abstract the source code according to Section III-A. Then, we compute the statement mask matrices for the two benchmark datasets, respectively. However, computing the matrices for a large amount of code is time consuming and inefficient, so we convert the computation into a series of matrix operations to fully use the computing resources of GPU and improve the efficiency (Section IV-C shows details). We test the time cost of the matrix computation before and after the acceleration, respectively. The results on the training set of GitProjs show that it reduces the computation time from 10 min to 7 s, indicating the efficiency of the acceleration operation.

Algorithm 1: Computation of the Statement Mask Matrix.

Input: $X = (x_1, x_2, \ldots, x_n) \in R^n$, which is a sequence of source code tokens.

Output: the statement mask matrix, $W^L \in R^{n \times n}$

1: function ComputeSMXM
2: \hspace{1em} $I \leftarrow \text{find the identifiers from } X$
3: \hspace{1em} // $W^A$ is lower triangular matrix, $W^M \in R^{n \times n}$
4: \hspace{1em} $W^A \leftarrow \text{lower triangular matrix}$
5: \hspace{1em} $W^A \leftarrow I(W^M)$
6: \hspace{1em} // repeat $W^A$ n times to get $W^B \in R^{n \times n}$
7: \hspace{1em} $W^B \leftarrow \text{repeat}(W^A)$
8: \hspace{1em} $W^B \leftarrow (W^B)^T$
9: \hspace{1em} $W^S1 \leftarrow |W^B - W^B| - |W^B - W^B|$
10: \hspace{1em} $W^S2 \leftarrow |W^B - W^B| - |W^B - W^B|$
11: \hspace{1em} $W^S \leftarrow (W^S1 + W^S2)/2$
12: return $W^S$

2) Hyperparameter Setting: DTrans is composed of six hidden layers and eight heads. The hidden layer size of the model and the size of every head are defined as 512 and 64, respectively. We train DTrans using the Adam optimizer [51] with an initial learning rate of 1.0 and use warm-up [52] to optimize the learning rate. We set the mini-batch size as 32 and the dropout as 0.1 during training. DTrans is trained for a maximum of 20 000 steps and performed early stops if the validation performance does not improve during 2000 steps. We also use beam search during inference and set the beam size as 10.

3) Platform: Our experiments are conducted on a single Tesla p100 GPU for about 10 h for $M_{\text{medium}}$ datasets and 5 h for $M_{\text{small}}$ datasets for both the benchmark datasets, respectively.

C. GPU Acceleration

Algorithm 1 shows how we use matrix operations to replace inefficient nested loops during computing the statement mask matrix.

The input $X = x_1, x_2, \ldots, x_n \in R^n$ is the sequence of source code tokens. We first compute $I = i_1, i_2, \ldots, i_n \in R^n$, which is a vector consisting of 0 and 1, from $X$. The rule for generating $i$: if $x_m \in X$ is an identifier, the value of $i_m \in I$ is 1; otherwise, it is 0 (line 2). We can get $W^A \in R^n$ by multiplying $I$ and the lower triangular matrix $W^M (W^M \in R^{n \times n})$ (lines 3 and 4). Next, we will repeat $W^A$ to get $W^B \in R^{n \times n}$ (line 5) and can find that if $i$ and $j$ are in the same statement, $W^B_{ij} = W^B_{ji}$, and vice versa. Therefore, finally, if $W^B_{ij} = W^B_{ji}$, we let $W^B_{ij} = W^B_{ji} = 1$; otherwise, it is $W^B_{ij} = W^B_{ji} = 0$. In this step (lines 6–10), we also use the matrix operations completely instead of nested loops, so this step is also efficient.

D. Evaluation Metrics

We evaluate the performance of DTrans in code editing using three popular metrics, including Exact Match [19], [32], BLEU-4 (bilingual evaluation understudy in 4-gram) [53], and ROUGE-L (recall-oriented understudy for gisting evaluation in longest common subsequence) [54].
Exact Match computes the number and percentage of predicted code changes that exactly match the changed code in the test sets.

BLEU-4 is a widely used metric in natural language processing and software engineering fields to evaluate the quality of generated texts, e.g., machine translation, code commit message generation, and code commit message generation [53, 55, 56]. It computes the frequencies of the co-occurrence of n-grams between the ground truth $\hat{y}$ and the generated sequence $y$ to judge the similarity

$$\text{BLEU-N} = b(y, \hat{y}) \cdot \exp \left( \sum_{n=1}^{N} \beta_n \log p_n(y, \hat{y}) \right)$$

where $b(y, \hat{y})$ indicates the brevity penalty, and $p_n(y, \hat{y})$ and $\beta_n$ represent the geometric average of the modified n-gram precision and the weighting parameter, respectively. We use corpus-level BLEU-4, i.e., $N = 4$ for evaluation since it is demonstrated to be more correlated with human judgments than other evaluation metrics [57].

ROUGE-L is commonly used in natural language translation [54] and is an F-measure based on the longest common subsequence (LCS) between candidate and target sequences, where the LCS is a set of words appearing in the two sequences in the same order

$$\text{ROUGE-L} = \frac{(1 + \beta^2) R_{\text{lcS}} P_{\text{lcS}}}{R_{\text{lcS}} + \beta^2 P_{\text{lcS}}}$$

where $R_{\text{lcS}} = \frac{\text{lcS}(X, Y)}{\text{len}(Y)}$ and $P_{\text{lcS}} = \frac{\text{lcS}(X, Y)}{\text{len}(X)}$. $X$ and $Y$ denote candidate sequence and reference sequence, respectively. LCS$(X, Y)$ represents the length of the LCS between $X$ and $Y$.

E. Comparison Model

We compare DTrans with three baseline models, including Tufano et al.’s (an NMT-based model) [19, 32], SequenceR [29], and CODIT [20]. Tufano et al. [19, 32] employ a typical encoder–decoder model long short-term memory (LSTM) to edit method-level code, where the input is a sequence of code tokens. SequenceR [29] is also an LSTM-based encoder–decoder model, but it uses copy mechanism to copy code tokens from the source code during decoding. The input of SequenceR is also code token sequence. CODIT [20] is a tree-based model, which uses the ASTs of source code as input and predicts code edit at the AST level.

V. EXPERIMENT RESULTS

In this section, we aim at verifying the effectiveness of the proposed approach, specifically by answering the following research questions.

RQ1: What is the performance of the proposed approach compared with the baseline models?

RQ2: What is the impact of the proposed dynamically relative position encoding on the model performance?

RQ3: What is the effectiveness of DTrans in generating multiline code change prediction?

RQ4: Whether DTrans can accurately locate the lines to edit for code change prediction?

RQ5: What is the impact of different parameters on the model performance?

RQ6: What is the performance of DTrans in cross-project setting?

Specifically, RQ1 is to evaluate the performance of the proposed model compared with baselines, including token-based models and tree-based models. To verify the advantage of the proposed dynamically relative position embedding, we compare DTrans with the Transformer [36] and the Transformer with relative position embedding (namely Transformer_rel) [40] in RQ2. For RQ3, since we find that more than 30% of the code samples in the datasets need multiline code changes, the research question is to evaluate the capacity of DTrans for generating multiple-line code changes. RQ4 is to validate the ability of locating lines to edit. Finally, since the hyperparameters can impact the performance of DTrans, RQ5 discusses the hyperparameter configurations. RQ6 is to evaluate the performance of DTrans in cross-project setting.

A. Answer to RQ1: Performance of the Proposed DTrans

1) Comparison With Token-Based Models: Table II presents the experimental results of our proposed model and the token-based baselines on the benchmark datasets. From the table, we can observe that DTrans performs better than the token-based baselines in predicting exact-matched code changes for all the datasets. For example, DTrans successfully generates 489 exact-matched code changes in Gerrit for $M_{\text{small}}$ and 409 for $M_{\text{medium}}$, while Tufano et al.’s only generates 388 and 334 exact-matched code changes, respectively, and SequenceR only generates 405 and 284 exact-matched code changes for $M_{\text{small}}$ and $M_{\text{medium}}$. Compared with Tufano et al.’s approach, DTrans outperforms 26.04% and 22.45% for $M_{\text{small}}$ and $M_{\text{medium}}$, respectively. Compared with SequenceR, DTrans outperforms 20.74% and 44.01% for $M_{\text{small}}$ and $M_{\text{medium}}$, respectively. For GitProjs, SequenceR outputs 2255 code changes that are consistent with the ground truth for $M_{\text{small}}$ and 1214 for $M_{\text{medium}}$, while DTrans successfully produces 2573 and 1625 for the two types of datasets, respectively. Besides, the ground truth is human-writing code [19, 28], so the higher scores of BLEU-4 and ROUGE-L represent that the results generated by DTrans are semantically similar to the human-writing code. For example, DTrans increases the performance of SequenceR by 2.59% and 0.66% in Google $M_{\text{medium}}$ with respect to the BLEU-4 and ROUGE-L metrics, respectively. The results demonstrate the effectiveness of the proposed DTrans over the token-based models.

2) Comparison With Tree-Based Models: Because CODIT does not provide the source code for data processing, and the data processing process of CODIT is very complex, we directly compare it on the code change dataset used by CODIT. Table III shows the experimental results of DTrans and CODIT. In the abstracted code change dataset provided by CODIT, the result of CODIT is not good. Compared with SequenceR, CODIT is lower than SequenceR by 17.80%, 4.59%, and 3.41% with respect to the Exact Match, BLEU-4, and ROUGE-L metrics, respectively.
DTrans improves the performance of SequenceR by 16.10%, 2.38%, and 0.83% regarding the three metrics, respectively.

**Answer to RQ1:** In summary, DTrans can more accurately predict code changes, and the generated code changes are more semantically relevant to the ground truth.

**B. Answer to RQ2: Impact of the Proposed Dynamically Relative Position Encoding on the Model Performance**

To evaluate the effectiveness of the proposed dynamically relative position encoding strategy, we compare DTrans with the original Transformer [36] and the Transformer with relative position (Transformer\textsubscript{relative}) [40]. We reproduce their experiments under the same hyperparameter settings as DTrans for fair comparison.

Table IV shows the experimental results; we find that the Transformer performs better than Tufano et al.’s approach. In more details, the Transformer improves the performance of Tufano et al. by 6.3–123.62%, 0.35–32.39%, and 0.15–9.64% regarding the three metrics on the two benchmark datasets, respectively. Besides, the Transformer can generate 4795 code changes that exactly match the ground truth on the two benchmark datasets, which outperforms SequenceR by 15.32%. The experimental results suggest that Transformer-based models can predict more effective code edits than token-based models. Moreover, Transformer\textsubscript{relative} performs better than the vanilla Transformer in most cases, which indicates that the relative position encoding in the Transformer is more effective in capturing the code edit patterns. Finally, DTrans achieves better performance than Transformer\textsubscript{relative}, with increase rates at 1.28% and 3.24% in terms of the exact match metric on the $M_{small}$ and $M_{medium}$ datasets, respectively. The results indicate the efficacy of the proposed dynamically relative position encoding strategy.

**Answer to RQ2:** In summary, the Transformer-based models outperform baselines. The statement-level syntactic information for position encoding facilitates more accurate code change prediction.
C. Answer to RQ3: Effectiveness of Code Change Prediction for Multiple Lines

According to the statistics [29], [49], most edits are accomplished through a single-line code change. However, some edits still need multiline code changes in practice. We analyze the multiline code changes in the Gerrit dataset [58] in Table V and observe that nearly 35% edits involve changes of more than one line of code.

We then investigate the effectiveness of DTrans in producing multiline code changes, with the evaluation results shown in Fig. 6. As illustrated in Table V, DTrans achieves the best performance among all the baselines in multiline code change prediction. For example, DTrans overall produces 42.25% exact-matched code changes for $M_{small}$ projects and 33.02% for $M_{medium}$ projects, while Tufano et al. only outputs 30.97% and 23.16% for the two types of datasets, respectively. The results indicate the usefulness of DTrans in multiline code prediction. We can also observe that compared with Tufano et al., the improvement of DTrans on the $M_{medium}$ projects (42.57%) is more significant than that on the $M_{small}$ projects (36.42%).

D. Answer to RQ4: Accuracy of DTrans in Locating Lines to Edit for Predicting Code Change

Locating correct lines to edit is the premise of the accurate code changes in the subsequent step. Therefore, in this research question, we analyze whether the proposed approach can accurately predict which lines to edit. Table VII shows the experimental results of locating the lines for editing. We can then analyze the average lines of code in the test sets of the $M_{small}$ and $M_{medium}$ projects, with the results shown in Table VI. We can find that the code in the $M_{medium}$ projects are longer than that in the $M_{small}$ projects on average. Therefore, we suppose that the significant performance of DTrans on the $M_{medium}$ projects may be attributed to that DTrans is more effective for predicting the changes of long code snippets than baseline models.

Answer to RQ3: In summary, DTrans demonstrates the superior ability of accurately generating multiple-line code changes and has a great improvement over the baselines.
observe that DTrans performs better than other techniques on all projects. For example, SequenceR can only locate 66.66% correct lines for M_{small} and 54.46% correct lines for M_{medium}, while DTrans can locate 70.28% and 59.01%, respectively. This observation demonstrates that DTrans can obtain more contextual information than other techniques.

Answer to RQ4: In summary, DTrans can greatly outperform the baselines in locating the lines to change (e.g., achieving 8.35% higher accuracy than the best baseline).

E. Answer to RQ5: Impact of the Model Parameters

In this section, we extend our experiments with different parameters to investigate the influence of internal factors of DTrans.

Fig. 7(a) presents the impact of the clipping distance \(k\) on the effectiveness of DTrans using other default configurations (defined in Section III-B). In this figure, the \(x\)-axis presents various clipping distances, while the \(y\)-axis presents the values of different evaluation metrics. We can find that the clipping distance does not impact the DTrans effectiveness much. For example, the largest performance difference among different clipping distances is within 2% for all the evaluation metrics. Since DTrans achieves good performance on the datasets when the clipping distance is 32, we choose the parameter as 32 during experimentation.

Fig. 7(b) presents the impact of the number of encoder–decoder block \(l\) on the effectiveness of DTrans using other default configurations (defined in Section II-B). In this figure, the \(x\)-axis presents different number of encoder–decoder blocks, while the \(y\)-axis presents the values of different evaluation metrics. From the figure, we observe that the number of encoder–decoder blocks has a significant impact on the model. For example, in GitProjs, the Exact Match of two-block DTrans is only 42.26%, while the Exact Match of six-block DTrans is 44.09%. Besides, more encoder–decoder blocks do not mean better performance. For example, six-block DTrans is better than the eight-block DTrans in Gerrit-All. Moreover, more encoder–decoder blocks will increase the model size and require more time to train. In terms of overall considerations, DTrans with six encoder–decoder blocks is a good option.

Answer to RQ5: In summary, the experimental results can be influenced by parameter configuration. Moreover, the clipping distance has little influence on DTrans, but the number of layers has much influence on DTrans.

F. Answer to RQ6: Performance of DTrans in Cross-Project Setting

In this section, we train the models in one project and test them in another project to simulate a more practical setting. We use the Gerrit dataset, which contains three different projects for the evaluation. We adopt the best Transformer-based baselines for comparison. The results are shown in Table VIII. We can observe that DTrans consistently performs better than Transformer and Transformer_{relative} in the cross-project setting. For example, when we train the models in Google M_{small} and then test them in Android M_{small}, DTrans can generate 17 exact-matched code changes, while the Transformer only generates 11 exact-matched code changes. Overall, DTrans increases the performance of the Transformer-based baselines by 0–200%, 0.03–5.66%, and 0.09–1.30% with respect to the Exact Match, BLEU-4, and ROUGE-L metrics, respectively. We can also find that despite the good performance of DTrans in cross-project setting, it presents obvious decline compared with the in-project performance, e.g., the exact match score drops by more than 80%. The Transformer-based baselines show the similar trend.
The phenomenon is reasonable since the edit patterns of different projects may be greatly different.

**Answer to RQ6:** In summary, DTrans performs better than the baseline models in the cross-project setting. However, the performance of all the models drops greatly compared with the in-project setting, indicating that cross-project evaluation is a more challenging setting for the code edit task.

## VI. CASE STUDY

To evaluate the performance of DTrans in predicting accurate code edits, we select three cases from benchmark datasets, as shown in Fig. 8.

Example (a) in Fig. 8 presents an example of code edit operation prediction. The method `addSlices` lacks an object to conduct the method function `slices.addAll(slices)`, so the correct edit operation is to add an object `this`. However, Tufano et al.’s approach mistakenly predicts the operation to change the return from `true` to `false`. Similarly, SequenceR incorrectly returns the variable `slices`. DTrans successfully predicts the correct edit operation and adds `this` to point to the variable inside the class.

In Example (b) in Fig. 8, the original code needs to remove `grade` from `curve`, but it does not check whether the variable `grade` is `null`. The correct edit operation is to check variable `grade` before executing `remove`. Tufano et al.’s approach does not check variable `grade` and just refactors original code. SequenceR predicts the correct operation method but the incorrect operation object. It checks `null` for `curve.remove(grade)` rather than variable `grade`. DTrans successfully predicts both the correct operation method and operation object. It checks whether the variable `grade` is `null` before executing `curve.remove(grade)`.

In addition, DTrans does not always predict accurate code changes. In Example (c) in Fig. 8, the original code needs a return statement because the modifier `void` does not appear in the method definition. Tufano et al.’s approach successfully adds `return` before `getCFlags()`. SequenceR mistakenly thinks that the original code should be a static function, so it inserts the modifier `static`. For DTrans, it successfully adds the `return` token, but incorrectly changes the API from `java.lang.Iterable` to `java.lang.Set`, which is a nonexistent interface. This example motivates us to create an API knowledge base to facilitate the code edit process in future.

## VII. THREAT TO VALIDITY

**Internal validity** is mainly about the hyperparameter configuration we adopted in our DTrans model. To reduce this threat, we conduct an experiment to study the impact of configuration, and we explain in Section V-E about how hyperparameters influence our model’s performance.

**Construct validity** is mainly the suitability of our evaluation metrics. To reduce this risk, we additionally introduce BLEU-4 [53] and ROUGE-L [54] to evaluate the effectiveness of our approaches, which can well simulate the nontrivial coding activities in evaluating generated code.

**External validity** is mainly concerned with whether the performance of our DTrans techniques can still be effective in other datasets. To reduce these threats, we additionally select 2.3 million pairwise code changes generated from GitHub open-source projects [32] to evaluate the effectiveness of our approach. And experimental results demonstrate the effectiveness of our approach (in Section V-A). To further reduce the threats, we are planning to collect more open-source projects to evaluate our approach. Besides, the quality of the datasets may be another threat. In this article, we simply follow the previous work [19], [20] by directly adopting the benchmark datasets without further cleaning. As illustrated in [59], high-quality datasets are very important for DL models. We will study the quality of the datasets in future work.

## VIII. RELATED WORK

Related works focus on two key aspects: position representations of the Transformer and automatic code edit.

### A. Position Representations of the Transformer

Unlike a recurrent neural network [60], which incorporates inductive bias by successively loading the input tokens, the
Transformer is less position sensitive [36]. It is critical to incorporate position encoding into the Transformer.

1) Absolute Position Representations: Vaswani et al. [36] proposed the Transformer and the trigonometric function to calculate positional information for each token, but the positional information cannot change, while Devlin et al. [61] and Liu et al. [62] use a parameter matrix to calculate positional information. Liu et al. [63] proposed FLOATER, which models position encoding as a continuous dynamical system and admit the standard sinusoidal position encoding as a special case, making more flexible in theory. Dehghani et al. [64] and Lan et al. [65] found that injecting the position information into
layers can improve the performance of the Transformer in some tasks.

2) Relative Position Representations: Relative position representations take the relative distance into calculating attention rather than absolute position, which performs more effective and flexible. Shaw et al. [40] first proposed the concept of relative position embedding and its application scope. Yang et al. [66] and Dai et al. [67] improved the relative position embedding to boost the effectiveness. Raffel et al. [68] and Ke et al. [69] evaluated the effective of “input-position” and “position-input” and remove them from the Transformer. He et al. [70] evaluated the absolute and relative position embedding and proved the usability of relation position embedding. Since these approaches are developed for natural language processing, they are unable to capture the statement-level information included in code, while our proposed dynamically relative position encoding strategy is specifically designed for involving the statement-level syntax information of source code.

B. Automatic Code Edit

Code edit throughout the program development and maintenance relates to various behaviors, e.g., automatic program repair [31], [71], [72], [73], [74], API-related update [26], and code refactoring [25], [75]. In recent years, more and more proposed works adapted DL techniques in automatic code edit [20], [24], [31], [76], aiming at automatically predicting code changes using a data-driven approach. Tufano et al. [19] applied NMT techniques to generate the target code at the method level. They treated code as natural language, converting it into tokens and using code abstraction to overcome the issue of OOV. Chen et al. [29] presented the SequenceR, an NMT-based approach, which uses the attention mechanism and outperforms Tufano et al. [19]. Chakraborty et al. [20] presented CODIT, a tree-based NMT model for predicting concrete source code changes and learning code change patterns in the wild, and it is the state-of-the-art NMT-based model in code edit. Above approaches ignore the statement-level information, so we propose DTrans, a novel Transformer-based approach, which explicitly incorporates the statement-level syntactic information for better capturing the local structure of code, to predict code changes.

IX. CONCLUSION

In this article, we introduced DTrans, a Transformer-based technique that can predict code changes from merged PR codes from developers. To better capture the statement-level information of code, DTrans was designed with dynamically relative position encoding in multilhead attention of Transformer. Compared with other DL-based techniques, such as NMT, DTrans can capture the syntactic information, which makes the generated code change higher quality. The experimental results showed that DTrans can more accurately generate program changes in automatic code edit.

Our experiments also demonstrated the difficulties in the cross-project code edit task. In the future, we plan to investigate the cross-project challenges and incorporate more semantic information (e.g., control-flow graph and abstract syntax tree) to increase our capacity in code editing for the cross-project task.

REFERENCES

[1] F. Ferreira, L. L. Silva, and M. T. Valente, “Software engineering meets deep learning: A mapping study,” in Proc. 36th Annu. ACM Symp. Appl. Comput., 2021, pp. 1542–1549.
[2] Y. Yang, X. Xia, D. Lo, and J. Grundy, “A survey on deep learning for software engineering,” 2020, arXiv:2011.14597.
[3] H. F. Eniser, S. Gerasimou, and A. Sen, “DeepFault: Fault localization for deep neural networks,” in Proc. Int. Conf. Fundam. Approaches Softw. Eng., 2019, pp. 171–191.
[4] X. Li, W. Li, Y. Zhang, and L. Zhang, “DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization,” in Proc. 20th ACM SIGSOFT Int. Symp. Softw. Testing Anal., 2019, pp. 169–180.
[5] M. Wardat, W. Le, and H. Rajan, “DeepLocalize: Fault localization for deep neural networks,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., 2021, pp. 251–262.
[6] T. Latelier, L. Pang, V. H. Pham, M. Wei, and L. Tan, “Encore: Ensemble learning using convolution neural machine translation for automatic program repair,” 2019, arXiv:1906.08691.
[7] Y. Li, S. Wang, and T. N. Nguyen, “DLFix: Context-based code transformation learning for automated program repair,” in Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., 2020, pp. 602–614.
[8] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “DeepFix: Fixing common C language errors by deep learning,” in Proc. AAAI Conf. Artif. Intell., 2017, vol. 31, no. 1, pp. 1345–1351.
[9] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” in Proc. 58th Annu. Meeting Assoc. Comput. Linguistics, 2020, pp. 4998–5007.
[10] L. LeClair, S. Haque, L. Wu, and C. McMillan, “Improved code summarization via a graph neural network,” in Proc. 28th Int. Conf. Prog. Comprehension, 2020, pp. 184–195.
[11] Y. Wan et al., “Improving automatic source code summarization via deep reinforcement learning,” in Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng., 2018, pp. 397–409.
[12] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., 2021, pp. 150–162.
[13] Y. Huang, X. Hu, N. Jia, X. Chen, Z. Zheng, and X. Luo, “CommntPst: Deep learning source code for commenting positions prediction,” J. Syst. Softw., vol. 170, 2020, Art. no. 110754.
[14] A. Hasanpour, P. Farzii, A. Tehrani, and R. Akbari, “Software defect prediction based on deep learning models: Performance study,” 2020, arXiv:2004.02589.
[15] J. Chen et al., “Software visualization and deep transfer learning for effective software defect prediction,” in Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., 2020, pp. 578–589.
[16] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep semantic feature learning for software defect prediction,” IEEE Trans. Softw. Eng., vol. 46, no. 12, pp. 1267–1293, Dec. 2020.
[17] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Conf. Symp. Found. Softw. Eng., 2019, pp. 510–520.
[18] M. Yen, R. Wu, and S.-C. Cheung, “How well do change sequences predict defects? Sequence learning from software changes,” IEEE Trans. Softw. Eng., vol. 46, no. 11, pp. 1155–1175, Nov. 2020.
[19] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshvanyk, “On learning meaningful code changes via neural machine translation,” in Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., 2019, pp. 25–36.
[20] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, “CODIT: Code editing with tree-based neural models,” IEEE Trans. Softw. Eng., vol. 48, no. 4, pp. 1385–1399, Apr. 2022.
[21] T. Latelier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “CoCoNut: Combining context-aware neural translation models using ensemble for program repair,” in Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal., 2020, pp. 101–114.
[22] N. Jiang, T. Latelier, and L. Tan, “CURE: Code-aware neural machine translation for automatic program repair,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., 2021, pp. 1161–1173.
[23] W. Tansey and E. Tilewich, “Annotation refactoring: Inferring upgrade transformations for legacy applications,” in Proc. 23rd ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl., 2008, pp. 295–312.
[24] N. Peng, L. Hua, M. Kim, and K. S. McKinley, “Does automated refactoring obviate systematic editing?,” in Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng., 2015, vol. 1, pp. 392–402.
[25] X. Ge, Q. L. DuBose, and E. Murphy-Hill, “Reconciling manual and automatic refactoring,” in Proc. 34th Int. Conf. Softw. Eng., 2012, pp. 211–221.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
