Integrating Tree Path in Transformer for Code Representation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Learning distributed representation of source code requires modelling the syntax and semantics of it. Recent state-of-the-art models leverage highly structured representations of source code, such as the syntax trees and paths therein. In this work, we propose the TPTrans, which encodes paths in the syntax tree and embeds them into the attention module of Transformers. Specifically, our methods encode the pairwise relative path between tokens of source code and the absolute path from the leaf node to the tree root for each token in the syntax tree. We draw inspiration from the ideas of positional encoding and modify them to incorporate these path encoding into the attentions of Transformers. Our approaches show the feasibility of embedding sequential information as an inductive bias for the attention module of Transformers. Extensive experiments and ablation studies on code summarization across four different languages demonstrate the effectiveness of our methods. Furthermore, we conduct the empirical analysis of the path encoding methods shown in previous research works. We release our code and trained models at `http://github.com/xxx`.

1 Introduction

Machine learning for source code aims at building models that learn semantically embedding of programs. The initial representation of source code relied on sequential models adopted from natural language processing, such as n-gram language model [Hindle et al., 2016, Hellendoorn and Devanbu, 2017], Recurrent Neural Networks (RNNs) [Wei et al., 2019] and so on. However, source code is more logical than natural languages, rich in structured information such as Abstract Syntax Tree (AST), data flow, and control flow. Therefore, these previous works struggle to capture the structural complexity of source code.

To model the structure in source code, some works leverage the AST of programs and linearize the graph by traversing it. [Alon et al., 2018] proposed code2seq, which represents source code as a set of pairwise paths over AST where each path is compressed to a vector using LSTMs [Hochreiter and Schmidhuber, 1997]. The code2seq obtains state-of-the-art for code representation using only AST information, demonstrating the effectiveness of path encoding. Besides, [Kim et al., 2020] leveraged another kind of path from the leaf node to the root of the AST by traversing up its ancestors and then coupled the representation of paths with the source code.

On the other hand, several works also leverage structured graphical models for modelling source code directly. [Allamanis et al., 2017] used Gated Graph Neural Network (GGNN) to embed the semantically meaningful relationships in the code. [Zhou et al., 2019] proposed a novel graph neural network with composite programming representation for vulnerability identification. However, graph neural networks typically rely on synchronous message passing, which requires iteration of message passing to aggregate information [Allamanis et al., 2017, Fernandes et al., 2018]. Besides,

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

primarily for computational reasons, the graph neural networks for programs usually compute few message-passing iterations. For these reasons, the representation of GNNs tends to be local and struggle to leverage long-range interactions. On the contrary, Transformer [Vaswani et al., 2017] allows global information flow at each step but lack the ability to model the structural complexity of source code. Recently, some works explore introducing the structural inductive bias in Transformers to access the global structural representation of source codes. [Hellendoorn et al., 2019] proposed the Graph Relation Embedding Attention Transformer (GREAT), which extends [Shaw et al., 2018] and bias traditional Transformers with relational information from graph edge types. After that, [Zügner et al., 2021] proposed the Code Transformer based on the backbone of XLNet in [Yang et al., 2019], which computes pairwise distances on the AST and integrate multiple different relations into the attention module of it.

In this paper, we pursue the research line of combining the Transformer-based model with additional structure information of source codes. Our first starting point is the structural model of code2seq [Alon et al., 2018], which obtains state-of-the-art of code summarization using only pairwise path information in AST. However, code2seq lacks the modelling of context, which leads us to explore combining the path representation with the source code context. The other starting point is the Code Transformer [Zügner et al., 2021], which counts the distances to capture the structure of the source code. However, the different nodes combinations in the path contain plenty of structure information, which is overlooked by encoding distances.

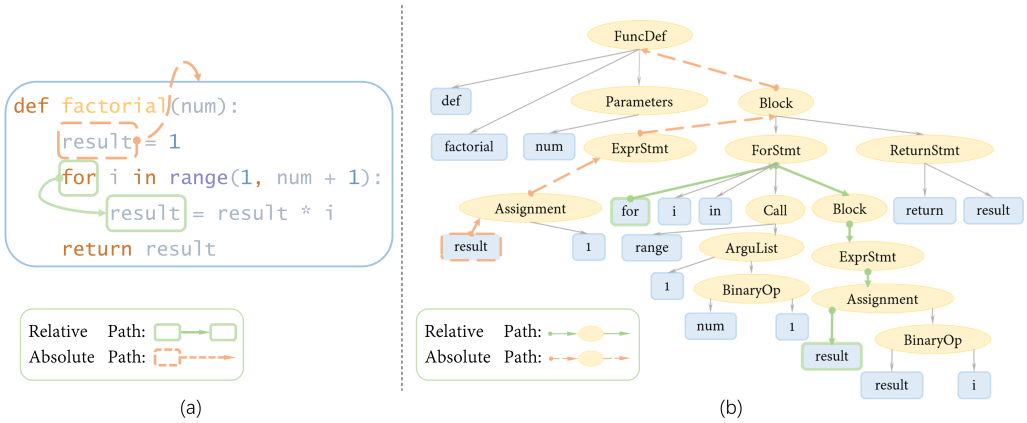

Figure 1: Example of (a) a python code snippet and (b) the concrete syntax tree of it. The relative path across the syntax tree between tokens **for** and **result** represents the relationship between them, showing the pattern of for loop in the code snippet. Meanwhile, the absolute path for token **result** starts at the leaf node and ends at the tree root, which reveals the behaviour of assignment for this token.

To overcome the drawback shown in previous works, we propose the TPTrans, which means integrating the **T**ree **P**ath into the **Trans**former. We adopt the ideas of path encoding to represent source code and integrate them to bias the attention of traditional Transformer. First of all, to combine path encoding with the context representation of tokens more naturally, we parse the code snippet into a concrete syntax tree (CST) instead of an AST. In a CST, each token of source codes has a unique match of the leaf node, which avoids manually mapping the nodes in the syntax tree to the tokens in [Zügner et al., 2021, Hellendoorn et al., 2019]. After that, we encode the path between tokens of source code across the syntax tree, and the path from leaf node to tree root for each token, respectively. For clarity, we name the first kind of path as the *relative path* and the last one as the *absolute path* (see Fig1).

Intuitively, the relative path represents the structural relationship between tokens and show the specific pattern of code block such as `For` loop, etc. Meanwhile, the absolute path complements the structural information for each token and reveal the program behaviour on it, such as `Assignment` and `Call`, etc. These kinds of paths introduce inductive bias into the attention module of the Transformer, powering it to know the structure of source codes.

We show the effectiveness of our model on the code summarization task on four datasets across different languages, in which the model predicts a method's name based on its body. Our model finally outperforms strong baselines and obtains state-of-the-art on most of all datasets. Besides, we also conduct additional experiments to analyze the relationship between these two kinds of paths. The contributions of this paper are summarized as follows:

1. We show the feasibility of embedding sequential information as an inductive bias for the attention module of Transformers and are the first ones to do so to our best knowledge.
2. We propose TPTRANS which integrates path encoding in Transformer, powers it to know the structure of source code, and our model significantly outperforms existing baselines.
3. We present the empirical study to investigate the relationship between two kinds of path encoding methods proposed in previous works [Alon et al., 2018, Kim et al., 2020], shedding light on future research on this line of work.

2 Related work

Representation Learning for source code The growing availability of open-source code creates opportunities for machine learning from big code. Early research works learned language model on raw text data for code [Dam et al., 2016, Wang et al., 2016, Allamanis et al., 2016, Iyer et al., 2016], providing evidence for the naturalness assumption proposed in [Hindle et al., 2016]. These models treat code as a sequence rather than a tree, struggle to model the structural complexity of source code. After that, some works leveraged the structural information to model source code. For example, [Mou et al., 2016] proposed tree-based convolutional neural networks to represent source code. [Alon et al., 2018, 2019] model source code as a weighted combination of pairwise paths in the abstract syntax tree. Several works also use structured graphical models to represent structural aspects of source code directly. [Allamanis et al., 2017] proposed GGNN to represent program graphs consisting of AST together with control-flow and data-flow edges. [Zhou et al., 2019] proposed to model graph-level classification for vulnerability identification of source code. After that, [Hellendoorn et al., 2019] and [Zügner et al., 2021] proposed to bias the self-attention computation of Transformer given the underlying graph structure. In this paper, we explore integrating path representation of syntax tree into Transformer, powering it to capture the structure of code snippets.

Structural representation of languages It has been proved that modelling the structure of the language helps to represent the compositional aspects of natural language [Socher et al., 2011] and improves the generalization of models [Chen et al., 2016]. In this way, several works [Shen et al., 2018, Shiv and Quirk, 2019] proposed new architectures for modelling tree structures, including in the domain of code representation learning [Fernandes et al., 2018, Zügner et al., 2021, Hellendoorn et al., 2019] and natural languages processing [Wang et al., 2019]. We pursue this line of research and show the benefits of integrating tree structure as an inductive bias.

Positional encoding for Transformer The absolute positional encoding used in Transformer has been proved not efficient to model the order information of sequence. [Shaw et al., 2018] firstly proposed to leverage relative positional encoding to Transformer. After that, Transformer-XL [Dai et al., 2019] re-parameterize the self attention of Transformer to integrate relative positional encoding and T5 [Raffel et al., 2019] simplified the vector representation of relative positions in [Shaw et al., 2018] to scalars. Recently, several works also try to enhance the positional encoding in Transformer to integrate structural inductive bias. For example, [Shiv and Quirk, 2019] extended the sequential positional encoding to tree-based positional encoding in Transformer and [Zügner et al., 2021] modify the relative positional encoding of Transformer-XL to incorporate multiple relations in tree structural. In this paper, we adopt the idea to modify positional encodings of Transformer and introduce structural bias of path information in the syntax tree for the self-attention computing.

3 Approaches

Self-attention is one of the key components in the Transformer model. The attention module can be formulated as querying a dictionary with key-value pairs. The matrix form equation for attention

with a single head is

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V, \tag{1}$$

where d is the dimensionality of the hidden representations, and Q(Query), K(Key), V(Value) are specified as the hidden representations of the previous layer. For simplicity, we omit the index of layer, denote $x = (x_1, x_2 \cdots, x_n)$ and $z = (z_1, z_2 \cdots, z_n)$ as the input and the output of the self-attention module in the same layer respectively, where n is the length of the sequence. Then Eq.1 can be rewritten as

$$z_i = \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{j'=1}^{n} \exp(\alpha_{ij'})}(x_j W^V),\ where\ \alpha_{ij} = \frac{1}{\sqrt{d}}(x_i W^Q)(x_j W^K)^T, \tag{2}$$

where $W^Q \in \mathbb{R}^{d \times d_v}$, $W^K \in \mathbb{R}^{d \times d_v}$, $W^V \in \mathbb{R}^{d \times d_v}$ and we set $d_k = d_v = d$. The Eq.2 computes all dot product of Query and Key equally, viewing them as unordered sets of vectors. That means the model is oblivious to structured inputs such as text or graphs. In NLP, several works [Shaw et al., 2018, Ke et al., 2020, Raffel et al., 2019, Dai et al., 2019] proposed different ways to bias Transformer towards sequential inputs by incorporating positional information into the self-attention module. Generally, there are two categories of positional encodings in the self-attention module: absolute positional encoding and relative positional encoding.

The original Transformer [Vaswani et al., 2017] proposed to use absolute positional encoding to represent positions. Given a sentence, a real-valued vector $p_i \in \mathbb{R}^d$ will be added to the word embedding w_i at position i, and the sum vector will be used as the input to the Transformer. In [Ke et al., 2020], the authors think there is no strong correlation between the word and absolute position. So they propose TUPE [Ke et al., 2020] to untie the correlations between absolute positions and words, and propose to directly model the relationships between a pair of words or positions by using different projection matrices, that is

$$\alpha_{ij} = \frac{1}{\sqrt{2d}}(x_i W^Q)(x_j W^K)^T + \frac{1}{\sqrt{2d}}(p_i U^Q)(p_j U^K)^T, \tag{3}$$

where $U^Q, U^K \in \mathbb{R}^{d \times d}$ are the projection matrices for the positional embedding and scaling term $\frac{1}{\sqrt{2d}}$ is used to retain the magnitude of α_{ij}.

Using different p_i for different position i helps the Transformer distinguish the words at different positions. However, the absolute positional encoding is not effective for the model to capture the relative word orders. Therefore, [Shaw et al., 2018] proposes a relative positional encoding to help the learning of the self-attention module, that is

$$z_i = \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{j'=1}^{n} \exp(\alpha_{ij'})}(x_j W^V + a_{ij}^V),\ where\ \alpha_{ij} = \frac{1}{\sqrt{d}}(x_i W^Q)(x_j W^K + a_{ij}^K)^T \tag{4}$$

where $a_{ij}^V, a_{ij}^K \in \mathbb{R}^d$ are learnable parameters and can be viewed as the embedding of the relative position between position i and j.

We draw inspiration from the ideas of encoding absolute and relative positional information and modify the formulations of [Ke et al., 2020] and [Shaw et al., 2018] to incorporate path information in the syntax tree of source code.

3.1 Integrating path encodings in the attention module

A syntax tree uniquely represents a source code snippet in a given language and grammar. Previous works [Alon et al., 2018, 2019, Kim et al., 2020] leverage paths in the syntax tree to represent the code snippet itself. In a syntax tree, the leaves are called terminals, and the non-leaf nodes are called nonterminals. [Alon et al., 2019, 2018] use Abstract Syntax Tree (AST) to represent a source code snippet. They consider pairwise paths between terminals, represent them as sequences for terminal and nonterminal nodes in AST, and then use these paths to represent the code snippet itself. [Kim et al., 2020] use another kind of path from the terminal to the root of the AST by traversing up its ancestors. They coupled the representation of paths with the terminals, then fed the sum representation into the Transformer.

We adopt the above methods of leveraging paths and integrate them into the attention module of Transformer. For clarity, we name the path between terminals as the relative path and the path from terminal to root as the absolute path. Unlike the above methods, We use the Concrete Syntax Tree (CST) instead of the AST to represent code snippets. In a CST, the terminals are all the tokens that appeared in the source code, so we take all of the tokens as the Transformer's input. In this way, we can naturally assign the relative paths between tokens and the absolute paths for each token, respectively. All kinds of paths are composed of nonterminals from a limited vocabulary of syntax grammar, and we use a learned embedding matrix to convert nonterminals to vectors.

Relative path encoding We name the path between terminals as the relative path since it represents the structural relationship between source code tokens. The embed vectors of a relative path are first fed into a sequence encoder. We use a bi-directional GRU [Cho et al., 2014] to encode the path and use the final state to represent it, that is

$$r_{ij} = GRU_r(Path_{x_i \to x_j}), \tag{5}$$

where $Path_{x_i \to x_j}$ is the sequence of embed vectors of nonterminals in the relative path between source code tokens x_i and x_j, and $r_{ij} \in \mathbb{R}^d$ is the final vector representation of this path. Then we modify the Eq.4 to integrate r_{ij} into the self-attention module. Firstly, we integrate r_{ij} into the Query-Key product, that is

$$\alpha_{ij} = \frac{1}{\sqrt{d}}[(x_i W^Q)(x_j W^K + \underbrace{r_{ij} W_r^K}_{1})^T], \tag{6}$$

and then we integrate r_{ij} into the weighted sum of Value, that is

$$z_i = \sum_{j=1}^{n} \frac{\exp \alpha_{ij}}{\sum_{j'=1}^{n} \exp \alpha_{ij}} (x_j W^V + \underbrace{r_{ij} W_r^V}_{2}), \tag{7}$$

where $W_r^K, W_r^V \in \mathbb{R}^{d \times d}$ are projection matrices of Key and Value for the relative path representation, respectively. For clarity of ablation study, we label the linear product of Key and Value, 1 and 2, respectively.

Absolute path encoding We name the path from terminal to tree root as the absolute path since it calibrates the source code token in the structure of programs. The embed vectors of an absolute path are fed into another bi-directional GRU [Cho et al., 2014], that is

$$a_i = GRU_a(Path_{x_i \to root}), \tag{8}$$

where $Path_{x_i \to root}$ is the sequence of embed vectors of nonterminals in the absolute path from the tokens x_i to the CST tree root, and $a_i \in \mathbb{R}^d$ is the final vector representation of this path. We modify the Eq.3 to integrate a_i into the self-attention module, that is

$$\alpha_{ij} = \frac{1}{\sqrt{d}}[(x_i W^Q)(x_j W^K)^T + (a_i W_a^Q)(a_j W_a^K)^T], \tag{9}$$

where $W_a^Q, W_a^K \in \mathbb{R}^{d \times d}$ are projection matrices of Query and Key for the absolute path representation, respectively[1]. To combine the relative path encoding shown in the previous paragraph further, we modify the Eq.9 as

$$\alpha_{ij} = \frac{1}{\sqrt{d}}[(x_i W^Q)(x_j W^K + \underbrace{r_{ij} W_r^K}_{1})^T + (a_i W_a^K)(a_j W_a^V)^T]. \tag{10}$$

In the two paragraphs above, we propose several modifications to integrate paths of the syntax tree into the attention module. To sum up, we combine them and propose two versions of models. We firstly name our method of relative path encoding as TPTRANS (Eq.6 + Eq.7) as our main proposed model for comparison with baselines, and the result is shown in subsection 5.1. Then, to further analyze the relationship between the relative path encoding and the absolute path encoding, we name the method of combining both relative and absolute path encoding as TPTRANS-α (Eq.10 + Eq.7) and show the result in the subsection 5.2.

[1]We tried to use scaling term $\sqrt{2d}$ in TUPE [Ke et al., 2020], but the experiment results indicate not improve much. Besides, we also tried the method adding the representation of path with terminal directly in [Kim et al., 2020] but no improvement, which we think is similar to the conclusion of untying correlations in TUPE.

4 Experiment setup

Code summarization is one of the most popular tasks in representation learning of source code. Given the body of a function, this task is to predict the name of the function. As observed by [Alon et al., 2019] and [Allamanis et al., 2016], code summarization is a good benchmark for code representation learning, as the method name tends to be descriptive and precise, and the method body typically forms complete logical units. In this task, our proposed model jointly learns both contexts of code tokens and the path representation in the syntax tree and then predicts the target method name as a sequence of sub-tokens. We adopt the measure used by [Alon et al., 2018, Zügner et al., 2021], who measured precision, recall, and F1 score over the target sequence in the case insensitive.

	Samples per partition		
Dataset	Train	Valid	Test
CSN-Python	412,178	23,107	22,176
CSN-Javascript	123,889	8,253	6,483
CSN-Ruby	48,791	2,209	2,279
CSN-Go	317,832	14,242	14,291

Table 1: Dataset statistics.

Datasets To show the effectiveness of our approaches across different languages of source code, we experiment on four datasets introduced in the CodeSearchNet (CSN) Challenge [Husain et al., 2019]: Python, Javascript, Go, and Ruby. The datasets from CodeSearchNet have been carefully deduplicated to avoid data leakage from the training set. See Table4 for a summary of the datasets we use in this work.

Preprocessing We produce a CST for each method using the open-source CST parser Tree-Sitter[2]. We split the token of terminals into sub-tokens, traversing from sub-token to get the relative path and the absolute path in the CST. We assign ids for all kind of nonterminals and do not split the literal of them. We set the maximum path length 32, make padding for short paths, and equal interval sampling for long paths. We adopt the way to split sub-tokens following [Zügner et al., 2021] and [Alon et al., 2018], in which each token of the source code is split into sub-tokens respective to code naming conventions, e.g., `setMaxConnectionsPerServer` is split into [`set`, `max`, `connections`, `per`, `server`]. We limit the vocabulary to sub-tokens with at least 100 occurrences in the training set, remove all the punctuation in the code sequence and restrict the maximum sequence length 512. We refer the reader to the appendix for further details.

Parameter sharing In our implementation, the path embedding matrix for nonterminals is shared for both absolute and relative paths while the path encoders are not. Since the number of nonterminal types is far less than the size of the vocabulary of source code, we set different embedding matrices for nonterminals and source tokens, respectively. For efficiency, we share in different layers and heads for the projection matrices W_r^K, W_r^V in relative path encoding and W_a^Q, W_a^K in absolute path encoding.

Efficiency A naive way to get path representation in the relative path encoding requires encoding $N \times N$ pairwise paths between any two sub-tokens. Besides, the naive way to compute α_{ij} in the attention module requires computing the Query-Key product between token and path for all pairs (i, j). To optimize them, we notice that there exists much repetition in all paths of a sample[3]. That means we can maintain a set $\mathbb{P}$ of unique paths, and a $N \times N$ matrix M to map the ids of paths in the set $\mathbb{P}$ to 2-dim coordinates for each sample, where N is the length of sequences. After that, we encode paths in the path set, compute the Query-Key product between each token and the path set, and map the attention scores to correct locations using matrix M. Besides, we also reduce the cost of the weighted sum of Value on the same principle. Finally, we reduce the costs to be linear. We refer the reader to the appendix for further details. In our experiment, we set the maximum size 512 for the set of relative path and 256 for the absolute path, which covers most of the samples.

[2]https://github.com/tree-sitter/

[3]For example, if we parse a function that has three formal arguments to a CST, we could see that the pairwise relative path between any two arguments will be the same.

Pointer network We add a pointer network [Vinyals et al., 2015] to the decoders of our models. Concretely, we use the pointer network equipped with sentinel [Merity et al., 2016] and the scaled dot-product attention to calculate the probability of pointing. The pointer network enhances predictions by pointing at positions in the input sequence and often improves results for less frequent tokens. It even enables the model to predict tokens not in vocabulary by pointing their positions in input.

Hyperparameters For all our experiments, we use Transformer with three layers, eight heads, and the activation function of GELU. We set the d and d_{FF} of Transformer to 1024 and 4096, respectively. The embedding dimensions of the word and path node are 512 and 64, respectively. We apply a linear layer to project word embedding to Transform's hidden dimension. We use one layer Bi-GRU equipped with layer normalization, set the hidden size to 64, and concat the final states of forward and backward as output. We set the mini-batch size and dropout rate to 128 and 0.2, respectively. We also employ label smoothing of 0.1. As the optimizer, we use Adam [Kingma and Ba, 2014] with a learning rate and weight decay of $1e^{-4}$. Besides, we use the learning rate scheduler `ReduceLROnPlateau` in Pytorch [Paszke et al., 2019], reduce the rate to $1e^{-5}$ when the F1 score does not get improvement. In our experiment, we use 4 Tesla V100 GPUs for training.

Baselines We mainly compare with code2seq [Alon et al., 2018], the Graph Relational Embedding Attention Transformer (GREAT) [Hellendoorn et al., 2019] and the Code Transformer [Zügner et al., 2021]. Code2seq is a non-Transformer model and state-of-the-art of code summarization using only pairwise path information in AST. GREAT uses the relative positional encoding framework present in [Shaw et al., 2018] to bias the attention via edges. Code Transformer is a recent Transformer model based XLNet [Yang et al., 2019], incorporating multiple different relations to leverage both structure and context jointly. Different to calculates multiple distances in Code Transformer, we embed the node in the syntax tree and encode the node sequence to represent the path relationship for structural representation of source code. In addition, since the specialized preprocessing using by GREAT is not public, we adopt the realize proposed in [Zügner et al., 2021] for GREAT. We also adopt the realize of XLNet in [Zügner et al., 2021] for comparison, which plus encoding of the token type returned by the tokenizer.

5 Results

5.1 Overall comparison

Table 2: Code summarization result on the CSN dataset.

Model	Python			Ruby			Javascript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Code2seq	35.79	24.85	29.34	23.23	10.31	14.28	30.18	19.88	23.97	52.30	43.43	47.45
Great	35.09	31.62	33.26	24.66	22.25	23.39	31.25	26.87	28.89	50.02	46.52	48.21
XLNet	37.39	32.01	34.49	29.88	25.89	27.74	33.33	26.86	29.75	51.79	47.58	49.60
Code Transformer	36.41	33.68	34.99	31.46	24.50	27.55	**35.07**	**29.65**	**32.13**	**55.09**	48.05	51.33
TPTrans	**39.32**	**34.25**	**36.61**	**33.14**	**27.87**	**30.28**	34.11	28.26	30.91	54.07	**51.64**	**52.83**

The overall comparison results are shown in Table 2. Our proposed model TPTRANS substantially outperforms all baselines on all but one language, highlighting the effectiveness of learning the path information in a syntax tree. The only exception is Javascript while our model still outperforms the most of baselines on this dataset. We investigate this exception further in the following ablation study 5.2 and conclude that since the structure of Javascript is much weak, the benefit of the encoding path becomes less significant compared to count distance.

It is also worth mentioning that the Code Transformer is a Transformer-XL-based model equipped with the relative positional encoding of context, which is effective for modelling long sequences. In this work, we mainly focus on studying the effectiveness of path embedding rather than the position encoding for context. For this purpose, We use the original Transformer equipped with absolute position encoding as our base model. Even so, we still obtain advanced results on almost all datasets, demonstrating the effectiveness of our method.

5.2 Ablation study

Table 3: Ablation study on the CSN dataset.

Model	Python			Ruby			Javascript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
TPTrans	**37.42**	**34.70**	**36.00**	**29.47**	**28.47**	**28.96**	**31.39**	**28.46**	**29.85**	**54.79**	**52.48**	**53.61**
w/o relative path in V	34.05	30.55	32.20	26.45	24.32	25.34	31.18	25.01	27.75	53.19	50.38	51.75
w/o relative path in K, V	28.39	24.93	26.55	23.14	19.80	21.34	28.09	23.41	25.54	50.88	47.16	48.95
TPTrans-α	**36.83**	**34.31**	**35.52**	**27.79**	**27.25**	**27.52**	**31.66**	**27.66**	**29.52**	**54.67**	**52.67**	**53.65**
w/o relative path in V	34.39	30.83	32.51	27.42	24.72	26.00	30.35	25.60	27.77	53.36	50.41	51.84
w/o relative path in K, V	32.80	28.32	30.40	25.79	23.10	24.37	29.36	25.54	27.32	53.41	49.01	51.12

In this work, we propose two versions of models, namely TPTRANS and TPTRANS-α. The first one encodes the pairwise relative paths in the syntax tree, while the last one additionally includes the absolute path for each sub-token. To analyze the effectiveness of the two kinds of paths respectively and the relationship between them, we remove the pointer network of all models and conduct the following ablation experiments shown in Table 3.

Relative path encoding In the top half of table 3, we show the experiment result of ablation study for TPTrans (Eq.6 + Eq.7). We first remove the relative path encoding in the weighted sum of Value (2 in Eq.7) and then further remove the relative path encoding in the Query-Key product (1 in Eq.6). We find that with the elimination of relative path encoding gradually, the performance of the TPTrans decreases obviously. After removing all submodules of relative path encoding (1 in Eq.6 and 2 in Eq.7), the model degenerates to the original Transformer in [Vaswani et al., 2017] and performs the worst (shown at the last row of the top half of Table 3). These results highlight the usefulness of relative path encoding, both in Query-Key product (Eq.6) and the weighted sum of Value (Eq.7).

Absolute path encoding Comparing to TPTrans, TPTrans-α extra adds absolute path encoding on the computing of attention score (Eq.10). To first analyze the effectiveness of the absolute path encoding exclusively, we remove all the relative path encoding of TPTrans-α and compare it to the original Transformer. The results are shown in the last rows of both top and bottom halves of Table 3, respectively. We find that the absolute path encoding significantly improves the original Transformer's performance, highlighting its effectiveness.

And then, we analyze the relationship between relative path encoding and absolute path encoding. In the bottom half of Table 3, We firstly find that with the introduction of relative path encoding gradually, the performance of TPTrans-α increases at the same time. This phenomenon is consistent with the ablation study for relative path encoding shown in the previous paragraph. Comparing the top and bottom halves of Table 3 row by row, we then find that with the introduction of relative path gradually, the improvement brought by the absolute path encoding decrease at the same time. Specifically, comparing the 2nd rows in both top and bottom halves of Table 3, we find that with the initial introduction of relative paths encoding in the Query-Key product (1 in Eq.10), the improvement brought by absolute paths encoding still exists slightly. Unfortunately, comparing the full model of TPTrans and TPTrans-α, we find that after introducing full of the relative path encoding, the improvement coming from the absolute path encoding diminishes entirely, and the performance even gets slight hurt on some datasets[4].

To the fact that the improvement of absolute path encoding fades gradually, we think that there indeed exists redundancy and even conflict between the relative and absolute path in some cases. If two sub-tokens are too far away from each other in the syntax tree, the relative path between them then spans the root of a tree, in which the relative and absolute path encoding play the same role for modelling structure. The experimental results also prove that it is better to encode the relative path relationship between tokens directly than to encode the absolute path for each token. For these reasons, we finally select the TPTRANS as our main proposed model based on these experiment results.

[4]We also tried to add absolute path representation to word embedding directly in [Kim et al., 2020] as mentioned in the subsection 3.1. As a result, the overall conclusion of improvement diminishing remains all the same.

Table 4: Ablation study on sequence encoding

Model	Python			Ruby			Javascript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
TPTrans	**37.42**	**34.70**	**36.00**	**29.47**	**28.47**	**28.96**	**31.39**	**28.46**	**29.85**	**54.79**	**52.48**	**53.61**
w/o relative path in V	34.05	30.55	32.20	26.45	24.32	25.34	31.18	25.01	27.75	53.19	50.38	51.75
degenerate path to hops												
TPTrans	**35.85**	**33.13**	**34.44**	**27.12**	**25.79**	**26.44**	**30.60**	**27.51**	**28.97**	**53.74**	**50.92**	**52.29**
w/o relative path in V	32.68	29.17	30.82	24.74	21.79	23.17	30.20	23.98	26.73	53.25	49.10	51.09

Encoding sequence as inductive bias To introduce structural bias into Transformer, previous works integrated embedding of either edge [Hellendoorn et al., 2019] or distance [Zügner et al., 2021] into the attention module of it. In this paper, we show the feasibility of embedding sequential information into self-attention computing. To verify the benefit of learning from paths, we conduct ablation studies: before feeding the path into Transformer, we first convert all nodes in the path to the same one. In such a case, what the model learns is degenerate completely into hops across the syntax tree between two terminals. We remove the pointer network of all models and show the ablation results in Table 4.

Comparing the top and bottom halves of Table 4 row by row, we find that after the path degenerates into hops, the models' performance shows noticeable declines, which indicates that our model indeed learns from rich context information of the path. These result also prove workable to encode sequential information as an inductive bias for the self-attention module. Besides, we can also find that in all four datasets, the decline in the Javascript dataset is not as evident as in others. We think it is because that the structure of Javascript is weaker than other languages, and therefore our model can not effectively benefit from paths on this dataset.

Table 5: Ablation study on pointer network

Model	Python			Ruby			Javascript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
With pointer network												
Code Transformer	36.41	33.68	34.99	31.46	24.50	27.55	**35.07**	**29.65**	**32.13**	**55.09**	48.05	51.33
TPTrans	**39.32**	**34.25**	**36.61**	**33.14**	**27.87**	**30.28**	34.11	28.26	30.91	54.07	**51.64**	**52.83**
Without pointer network												
Code Transformer	**37.74**	31.85	34.54	23.32	25.21	24.23	**33.15**	**28.72**	**30.78**	54.30	50.11	52.12
TPTrans	37.42	**34.70**	**36.00**	**29.47**	**28.47**	**28.96**	31.39	28.46	29.85	**54.79**	**52.48**	**53.61**

Pointer network In Table 5, we show the ablation results for the pointer network of TPTrans and Code Transformer. Comparing the results displayed at the top and bottom halves of Table 5, we find that the pointer network improves the performance of both TPTrans and Code Transformer on all but one language. We think the exception of the Go dataset is caused by the apparent violation of the i.i.d assumption, which has been analyzed in [Zügner et al., 2021] and we refer the reader to it for further details. Besides, we also find that our proposed model still outperforms the baseline on most of the datasets after removing the pointer network, especially gains 4.7 points improvement on Ruby dataset, shown the effectiveness of our method again.

6 Conclusion

We present TPTrans, which encodes paths in the syntax tree of a code snippet to represent structural information and integrate them into the attention module of Transformer. Specifically, we encode the pairwise relative path between tokens of source code and the absolute path from the leaf node to the root of the tree for each token and integrate them to bias the attention of Transformers. Our approaches show the effectiveness of integrating path sequential information into the self-attention of Transformer. Extensive experiments demonstrate that TPTrans achieves advanced performance on code summarization across four different programming languages. Besides, we further investigate the relationship between the absolute and relative path encoding, shedding light on future research.

References

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *arXiv preprint arXiv:1910.05923*, 2019.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*, 2020.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*, 2021.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.

Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719, 2016.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Richard Socher, Cliff Chiung-Yu Lin, Andrew Y Ng, and Christopher D Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

Qian Chen, Xiaodan Zhu, Zhenhua Ling, Si Wei, Hui Jiang, and Diana Inkpen. Enhanced lstm for natural language inference. *arXiv preprint arXiv:1609.06038*, 2016.

Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered neurons: Integrating tree structures into recurrent neural networks. *arXiv preprint arXiv:1810.09536*, 2018.

Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems*, 32:12081–12091, 2019.

Yau-Shian Wang, Hung-Yi Lee, and Yun-Nung Chen. Tree transformer: Integrating tree structures into self-attention. *arXiv preprint arXiv:1909.06639*, 2019.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Guolin Ke, Di He, and Tie-Yan Liu. Rethinking the positional encoding in language pre-training. *arXiv preprint arXiv:2006.15595*, 2020.

Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]