A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks

Abstract

We suggest two approaches to incorporate syntactic information into transformer models encoding trees (e.g. abstract syntax trees) and generating sequences. First, we use self-attention with relative position representations to consider structural relationships between nodes using a representation that encodes movements between any pair of nodes in the tree, and demonstrate how those movements can be computed efficiently on the fly. Second, we suggest an auxiliary loss enforcing the network to predict the lowest common ancestor of node pairs. We apply both methods to source code summarization tasks, where we outperform the state-of-the-art by up to 6% F1. On natural language machine translation, our models yield competitive results. We also consistently outperform sequence-based transformers, and demonstrate that our method yields representations that are more closely aligned with the AST structure.

1 Introduction

Modeling the semantics of source code has recently emerged as a research topic, with applications in duplicate detection (Baxter et al., 1998), automatic summarization (Alon et al., 2018a), natural language database querying (Xu et al., 2017), bug triage (Mani et al., 2019) and semantic code retrieval (Gu et al., 2018). We focus on source-code related sequence generation tasks such as code summarization. Here, the most successful models learn embeddings of code elements and put a strong focus on code structure, usually exploiting the abstract syntax tree (AST) (Baxter et al., 1998; Alon et al., 2019, 2018a). In contrast, transformer networks like BERT (Devlin et al., 2018) – which are currently considered state-of-the-art in modeling natural language – are weak at exploiting such structure: Their key component, self-attention, is based on a pairwise comparison of all tokens in the input sequence, whereas "structure" is only represented by adding absolute positional embeddings to the input.

To overcome these limitations, several approaches have been suggested recently. These linearize the input tree using a pre-order traversal and apply a modified "tree" transformer, either using absolute positional embeddings that encode structure (Shiv and Quirk, 2019), aggregation (Nguyen et al., 2020), or a boosting of attention weights with relative node positions (Kim et al., 2020). We continue this line of research by three contributions[1]:

1. We extend relative positional embeddings (Shaw et al., 2018) to encode structural relationships between nodes in trees. To do so, we demonstrate how relative positions for trees can be computed efficiently and densely during training using simple matrix operations.

2. While auxiliary losses – such as next sentence prediction (Devlin et al., 2018) or sentence reordering (Sun et al., 2019) – have already been shown to improve accuracy on sequence tasks, we suggest a structural loss for trees by predicting lowest common ancestors.

3. We explore the two above strategies in quantitative experiments on source code understanding and machine translation. Combining our two approaches outperforms the state-of-the-art by up to 6% and offers significant improvements over sequential transformer baselines. Compared to other recent work on tree transformers, our approach either performs better (Shiv and Quirk, 2019) or performs comparable while offering a much more scalable training (Nguyen et al., 2020). Finally, our approach requires only moderate adaptations to the conventional transformer architecture

[1] Our code and a demo are available under: `hidden`.

(namely, relative positional embeddings), and can be applied orthogonally to other tree extensions.

2 Related Work

Abstractive summarization condenses a source sequence into a short descriptive target sequence, while maintaining its meaning (Fan et al., 2017), and has been approached with encoder-decoder architectures ranging from recurrent neural networks (optionally with attention) (Bahdanau et al., 2014) over convolutional networks (Gehring et al., 2017) to the attention-based transformer (Vaswani et al., 2017) architectures. Many summarization approaches use pointer networks (Vinyals et al., 2015) to mitigate the out-of-vocabulary problem by copying words from the input sequence (See et al., 2017), but recently attention has shifted to Byte-Pair Encoding (BPE) (Sennrich et al., 2015), which we utilize in this work.

Modeling the semantics of source code by predicting precise summaries or missing identifiers has been extensively studied in the recent years (Allamanis et al., 2017). Allamanis et al. (2016) use a convolutional attention network over the tokens in a method to predict sub-tokens in method names. Recently, masked language model-based approaches (Devlin et al., 2018) been applied as well, for which Feng et al. (2020) train CodeBERT on pairs of natural language and methods. However, most models treat source-code as a token sequence and do not exploit additional structural information provided by existing syntax parsers.

There have been various approaches to utilize structural information: Tai et al. (2015) propose the TreeLSTM network, which recursively encodes a tree by computing a node's representation based on its children with an LSTM. The inverse problem of generating code from descriptions in natural language has been addressed by Yin and Neubig (2017) following a rule-based approach over ASTs. Similar to our work (Hu et al., 2018) linearize an AST and use a longer structure-based traversal (SBT) as input for a regular sequence-to-sequence model to predict comments. LeClair et al. (2019) summarize source-code by using two encoder networks, one that encodes the SBT of the AST and another the textual information in the sequence. For the same purpose Alon et al. (2018a)'s code2seq model encodes paths between terminal tokens in an AST using a dedicated encoder-decoder architecture with attention. In contrast code2seq, our model encodes the full AST with all relative positions at once and additionally and implicitly models the path between any two nodes. LeClair et al. (2020) and Fernandes et al. (2018) propose a structured summarization approach for code and natural language by adding a graph neural network on top of a sequence-to-sequence encoder. The above approaches utilize neither transformer networks, nor structural losses or relative position representations.

While only few of the above methods employ transformer encoders, our work is closest to recent approaches enabling transformers to utilize syntax trees. Shiv and Quirk (2019) define absolute positional embeddings for *regular* trees and show that these can be used to leverage syntactic information (by converting the tree into a binary tree). Our model processes non-regular trees without modification, and orthogonally shows that relative positional representations are also appropriate to represent trees in transformers. Kim et al. (2020) utilize pre-computed relative node positions similar to ours, but only apply them for a scalar boosting of attention weights during self-attention. The hierarchical transformer (Nguyen et al., 2020) uses aggregation, masking and hierarchical embeddings during self-attention to incorporate structure into transformers. In contrast to this, our approaches pose rather mild modifications (structural embeddings) or no modification at all (structural loss) to the standard transformer architecture, resulting in tolerable performance drops.

3 Approach

As illustrated in Figure 1, our approach extends transformer models (Vaswani et al., 2017) to incorporate structural information from trees. For source code, those trees are abstract syntax trees (ASTs), for natural language we use constituency parse trees. Specifically, we present two approaches: First, we extend relative position representations (Shaw et al., 2018) to encode the pairwise relations between nodes as movements (Section 3.1). Second, we introduce a structural loss enforcing the model to predict the lowest common ancestor of two nodes (Section 3.2) based on the encoder output.

We feed a tree into the transformer by replacing the conventional sequence of tokens with a pre-order sequence of nodes $\mathbf{x} = (x_1, x_2, x_3, \ldots, x_n)$

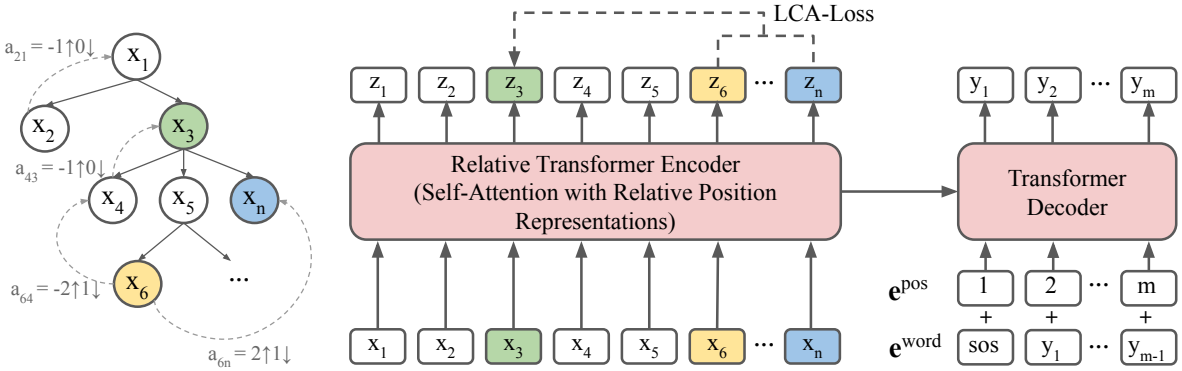


Figure 1: Our approach feeds a pre-order traversed tree (left) into a transformer encoder (middle). We use self-attention with relative position representations and encode relationships between all nodes by up- and down movements (gray dashed arrows). For example, the relation between x_6 (yellow) and x_n (blue) is $+2\uparrow 1\downarrow$ (2 steps up, then right (+), then 1 step down). Our structural loss is based on predicting the lowest common ancestor (green) between randomly sampled node pairs (yellow and blue). The decoder operates on absolute positional embeddings.

including both terminals and non-terminals. Every node i is reached via a unique path from the root $(1 =) i_1, i_2, \ldots, i_u\ (= i)$, where i_{l-1} is the parent of i_l and u (or $depth(i)$) denotes the depth of node i. Based on this path, we define the ancestors and descendants of Node i:

$$anc(i) := \{i_1, \ldots, i_u\}$$
$$desc(i) := \{j \neq i \mid i \in anc(j)\}.$$

Note that while the ancestors include i, the descendants do not. Finally, we define the lowest common ancestor (LCA) of nodes number i and j as

$$lca(i, j) = \underset{a \in anc(i) \cap anc(j)}{\arg\max} depth(a)$$

The node sequence $\mathbf{x}$ is first transformed into a sequence of real-valued input embeddings $\mathbf{x}_i \in \mathbb{R}^d$, which are processed by a regular transformer encoder (Vaswani et al., 2017) (for brevity we omit details here), resulting in a sequence of continuous representations $\mathbf{z}(\mathbf{x})$, or shorter $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_n)$. From this, a standard auto-regressive transformer decoder generates an output token sequence $\mathbf{y} = (y_1, \ldots, y_m)$, e.g. a summary of the input program $\mathbf{x}$. When generating token y_{i+1}, the decoder attends to the whole encoded sequence $\mathbf{z}$ as well as all previously generated symbols $y_1, \ldots, y_i$.

3.1 Relative Position Representations in Trees

Our model builts on relative position representations (RPR) (Shaw et al., 2018), which influence the attention between two tokens based on their pairwise relative position. A relative self-attention head in an transformer layer operates on an input sequence of embeddings $\mathbf{x}^L = (\mathbf{x}_1^L, \ldots, \mathbf{x}_n^L)$ with $\mathbf{x}_i^L \in \mathbb{R}^d$ and outputs a new sequence $\mathbf{x}^{L+1}$ with $\mathbf{x}_i^{L+1} \in \mathbb{R}^{d_h}$. The pairwise relationship between input elements $\mathbf{x}_i^L$ and $\mathbf{x}_j^L$ is represented by learned embeddings $\mathbf{a}_{ij} \in \mathbb{R}^{d_h}$ that are shared between attention heads, but not between layers. The output of self-attention with RPR is computed by

$$\mathbf{x}_i^{L+1} = \sum_{j=1}^{n} \alpha_{ij}(\mathbf{x}_j^L \mathbf{W}^V) \quad (1)$$

whereas the weight coefficient α_{ij} is computed by a softmax over compatibility scores e_{ij}:

$$e_{ij} = \frac{\mathbf{x}_i^L \mathbf{W}^Q (\mathbf{x}_j^L \mathbf{W}^K + \mathbf{a}_{ij})^\top}{\sqrt{d}} \quad (2)$$

where $d \times d_h$ matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ map the inputs to a head-specific embedding space[2]. Note that Equation (2) is the original self-attention from Vaswani et al. (2017) if one omits $\mathbf{a}_{ij}$.

While Shaw et al. (2018) define the relative positional embeddings $\mathbf{a}_{ij}$ based on the linear distance between sequential input tokens, we extend $\mathbf{a}_{ij}$ to take the input *tree* structure into account. We encode the path between nodes i and j in a matrix $\mathbf{M} \in \mathbb{N}^{n \times n}$: From i, we first take M_{ij} steps upward to $lca(i, j)$ and then M_{ji} steps downward to j. We investigate two options for representing this path:

[2]Note that Shaw et al. (2018) originally defined relative position representations $\mathbf{a}_{ij}^K, \mathbf{a}_{ij}^V$ for keys and values, but showed in experiments that key embeddings seem to suffice. Correspondingly, we only use key embeddings.

- the **path length** (clamped to a maximum) $l(i,j) = M_{ij}+M_{ji} \in \{0,...,C\}$. Then, $\mathbf{a}_{ij}$ is derived from an embedding table $\mathbf{E}$:

$$\mathbf{a}_{ij} := \mathbf{E}_{\mathbb{1}_{i<j}, l(i,j)} \tag{3}$$

  $\mathbb{1}_{i<j}$ indicates if Node i is left of Node j.

- **Movement pattern**: We distinguish between steps up (M_{ij}) and down (M_{ji}). Consider Figure 1, where we take 2 steps up and 1 step down from node x_6 to x_4 ($-2\uparrow 1\downarrow$). Both values are used as indices to the embedding table:

$$\mathbf{a}_{ij} := \mathbf{E}_{\mathbb{1}_{i<j}, M_{ij}, M_{ji}} \tag{4}$$

  We clamp both M_{ij} and M_{ji} to a maximum of C steps, such that the embedding table has $2\times(C+1)\times(C+1)$ entries.

Computing relative node positions efficiently The basis for relative position representations is the matrix $\mathbf{M}$. We show that $\mathbf{M}$ – and with it $\mathbf{a}_{ij}$ – can be derived efficiently with matrix operations: We first represent the tree using a binary *node incidence matrix* $\mathbf{N} \in \{0,1\}^{n\times n}$ (see Appendix A for an illustration) which encodes each node's path to the root by

$$N_{ij} = \begin{cases} 1 & \text{if } j \in anc(i) \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Note that we defined $i \in anc(i)$, and thus N_{ii}=1. Based on $\mathbf{N}$, we compute

- the depth of each node i by row-wise summation: $depth(i) = \sum_j N_{ij}$.

- a (symmetric) *ancestral matrix* $\mathbf{A} \in \mathbb{N}^{n\times n}$ (Andriantiana et al., 2018) whose entries A_{ij} contain the level of $lca(i,j)$ (or in other words the length of the common prefix) of two nodes i and j by

$$\mathbf{A} = \mathbf{N}\mathbf{N}^\top \tag{6}$$

- the matrix $\mathbf{M} \in \mathbb{N}^{n\times n}$ by

$$M_{ij} = depth(i) - A_{ij} \tag{7}$$

Note that – since in a pre-order traversal each node's descendants directly follow the node – we can easily derive $\mathbf{N}$ from the size of each node's sub-tree, by filling $|desc(j)|$+1 rows in column j with ones, starting at the diagonal ($N_{i,j}$=1 for $i = j, j+1, ..., j+|desc(j)|$). The number of descendants per node can be pre-computed in $O(n)$ time and space, and the above operations can be efficiently conducted on a batch of trees on a GPU using parallel matrix multiplications in $O(n^3)$, which we found feasible for input lengths commonly used in NLP (e.g., 1024 – see Figure 4 in the Appendix).

3.2 Structural Loss

The goal of our structural loss is to enforce the encoded representations $\mathbf{z}$ to adopt a notion of structural similarity from the AST. Thereby, we consider two nodes as "similar" if they are part of the same low-level syntactic unit (e.g., an if-statement), while nodes representing different passages in a text/program are considered dissimilar. We enforce this notion of similarity by training our model to predict nodes' LCAs. To do so, we concatenate the encoded representations $\mathbf{z}_i, \mathbf{z}_j$ of two nodes to predict their lowest common ancestor a:

$$\mathbf{v}_{ij} = RELU([\mathbf{z}_i; \mathbf{z}_j] \cdot \mathbf{W} + \mathbf{b})$$
$$p^{lca}(a|i,j) = softmax(\mathbf{v}_{ij} \cdot \mathbf{Z})_a$$

where $\mathbf{W} \in \mathbb{R}^{2d\times d}$ and $\mathbf{b} \in \mathbb{R}^d$ are learned, and $\mathbf{Z} \in \mathbb{R}^{d\times n}$ contains the stacked encoded representations. For each position a, the resulting softmax vector contains the probability $p^{lca}(a|i,j)$ that node a is node i and j's lowest common ancestor. We then use a log-likelihood loss over node pairs (i,j):

$$L_{lca}(\Theta) = -\sum_{i=1}^{n}\sum_{j=1}^{n} log\ p^{lca}(lca(i,j)|i,j,\mathbf{z}) \tag{8}$$

where Θ denotes the collection of model parameters and $\mathbf{z}$ is the encoder output. We complement this loss with a conventional cross entropy translation loss (Vaswani et al., 2017)

$$L_{trans}(\Theta) = -\sum_{i=1}^{n} log\ p(y_i|y_{\leq i-1}, \mathbf{x}) \tag{9}$$

More specifically, we use the label-smoothed version of Equation (9) (Szegedy et al., 2015). $(\mathbf{x}, \mathbf{y})$ denote training pairs consisting of pre-ordered input trees $\mathbf{x}$ and corresponding output sequences $\mathbf{y}$. Overall, we train the model by minimizing both losses simultaneously, i.e. our loss function is $L(\Theta) = L_{trans}(\Theta) + \gamma_{lca} \cdot L_{lca}(\Theta)$.

Sampling for Ancestor Prediction Instead of a dense loss computation (Equation (8)), we implement an efficient random sampling of M node pairs (i, j) with their lowest common ancestor $a = lca(i, j)$. Instead of sampling i and j and computing the corresponding a, we sample a and then draw random descendants (i, j) of a such that $a = lca(i, j)$. This can be done in $O(1)$ per sample: We first draw the LCA a, whereas the probability of drawing Node a is chosen proportionally to the number of its descendants. This gives more weight to nodes at the top of the tree (which are more likely to occur as LCAs). Given ancestor a, we sample two nodes i, j from a's descendants. Since the sequence is a pre-order traversal of the tree, a's descendants can be found at positions $a{+}1,\ldots,a{+}|desc(a)|$. We distinguish two cases: (1) If a has at least two children, we draw samples i, j from two different children (or their descendants). (2) If a has only a single child, we choose $i{=}a$ and j as a random descendant of a (if i and j were both descendants of a, their LCA would not be a but a's child or one of its descendants).

4 Experimental Setup

Most code-to-sequence models are evaluated on *method naming* and *code captioning* (or code-to-documentation) tasks (Alon et al., 2018a). Method naming aims to predict the usually short name of a method given its signature and body. Our second task – code captioning – aims at generating a longer natural language description of a given function, whereas the first line of a documentation is used as ground truth. On all code related tasks we use ASTs as input. Additionally, we show that our model is also applicable to natural language where we translate using constituency parse trees as input.

4.1 Preprocessing Source Code

When parsing code to ASTs[3], the set of terminal nodes becomes large. To overcome this problem, we follow common NLP practice (Babii et al., 2019) and tokenize the identifiers associated with terminal nodes into fine-grain tokens: (1) Inspired by Alon et al. (2018a), we split all identifiers on camel case or underscores, i.e. "`getNumber_Hex16`" will become "`get Number Hex 16`". (2) Next, we apply Byte Pair Encoding (BPE) on the above tokens (Sennrich et al., 2015), obtaining "`get Num@@ ber Hex 16`". (3) To reduce the vocabulary further, we modify the BPE algorithm so that when a token is a number, it is split into single digits, similar to character-level language modeling (Al-Rfou et al., 2019). This allows us to represent all possible numbers with only 20 tokens in our vocabulary. The example becomes "`get Num@@ ber Hex 1@@ 6`". (4) Additionally we replace all string and character literals in the source code with an identifier (e.g. `<STRING>`). Note that this tokenization yields multiple tokens for each terminal node/identifier, which alters the tree structure. When splitting a terminal node x_i into tokens $(t_1, \ldots, t_m)$, we replace x_i with these tokens. Each token becomes a new node, the first token t_1 with x_i's parent as parent and each other token with its predecessor as parent. In the example, we obtain `get`←`Num@@`←`ber`←`Hex`←`1@@`←`6`. Note that the BPE encoding is removed when computing evaluation measures such as BLEU.

[3] We use the tree-sitter library for parsing code into ASTs.

4.2 Tasks

Task 1: Method Naming We evaluate the method naming task on three different datasets introduced by Alon et al. (2018a): `java-small`, `java-med`, and `java-large`. All three datasets consist of java files from selected open-source projects, splitted into training, validation and test sets on project-level.

The `java-small` dataset consists of java files from 11 different projects, from which 9 are used for training, 1 is used for validation and 1 for testing. It contains around 700k samples/methods. The `java-med` dataset consists of 800 projects for training, 100 for validation and 100 for testing, containing about 4M samples. The largest dataset `java-large` contains 9,000 projects for training, 250 for validation and 300 for testing, containing $16M$ samples. Method names in all three datasets have 3.1 BPE tokens on average.

Following the work of Allamanis et al. (2016) and Alon et al. (2018a), we predict the method name as a sequence of sub-tokens splitted on camel case and underscores (compare Section 4.1). Note that the BPE encoding applied in preprocessing is removed before evaluating the model's performance. Like Alon et al. (2018a), we report case-insensitive micro precision, recall and F-measure over the target sequence.

Task 2: Code Captioning We evaluate the code captioning task on the `FunCom` dataset intro-

Model	java-small			java-med			java-large		
	P	R	F1	P	R	F1	P	R	F1
Paths+CRFs (Alon et al., 2018b)†	8.4	5.6	6.7	32.6	20.4	25.1	32.6	20.4	25.1
code2vec (Alon et al., 2019)†	18.5	18.7	18.6	38.1	28.3	32.5	48.2	38.4	42.7
TreeLSTM (Tai et al., 2015)†	40.0	31.8	35.5	53.1	41.7	46.7	60.3	48.3	53.6
2-layer BiLSTM†	42.6	30.0	35.2	55.2	41.8	47.5	63.5	48.8	55.2
ConvAttention (Allamanis et al., 2016)†	50.3	24.6	33.1	60.8	26.8	37.2	60.7	27.6	38.0
Transformer (no tree) (Alon et al., 2018a)†	38.1	26.7	31.4	50.1	35.0	41.2	59.1	40.6	48.1
code2seq (Alon et al., 2018a)†	50.6	37.4	43.0	61.2	47.1	53.2	64.0	55.0	59.2
Transformer (no tree) (Vaswani et al., 2017)	48.5	45.9	45.9	57.5	57.1	56.2	66.2	63.8	63.9
Absolute Tree Transformer ($k = 64$) (Shiv and Quirk, 2019)	47.2	45.6	45.0	59.3	57.9	57.3	66.6	64.2	64.3
Relative Structural Transformer	**52.7**	**47.6**	**48.6**	**61.3**	**60.0**	**59.4**	**66.6**	**64.6**	**64.5**

Table 1: Micro F1 for *Method naming*. Results marked with † are taken from Alon et al. (2018a).

Model	B1	B2	B3	B4	BLEU-4
attendgru (LeClair and McMillan, 2019)†	-	-	-	-	17.4
ast-attendgru (LeClair et al., 2019)‡	37.1	21.1	14.27	10.9	18.7
graph2seq (Xu et al., 2018)‡	37.6	21.3	14.1	10.6	18.6
code2seq (Alon et al., 2018a)‡	37.5	21.4	14.4	11.0	18.8
BiLSTM+GNN-LSTM (Fernandes et al., 2018)‡	37.7	21.5	14.6	11.1	19.1
code+gnn+BiLSTM-2hops (LeClair et al., 2020)‡	39.1	22.5	15.3	11.7	19.9
Transformer (no tree) (Vaswani et al., 2017)	40.3	23.6	16.4	12.6	21.1
Relative Structural Transformer	**42.3**	**24.4**	**16.8**	**12.9**	**21.7**

Table 2: *Code captioning* on the `FunCom` dataset (java). We report cumulative BLEU-4 score, together with single n-gram scores up to 4 n-grams (B1, ..., B4), evaluated with the script released along with the dataset. Results marked with † have been reported by LeClair and McMillan (2019), results marked with ‡ by LeClair et al. (2020).

duced by LeClair and McMillan (2019) and on the `CodeSearchNet` dataset (Husain et al., 2019).

The `FunCom` dataset consists of 2.1M java function/description pairs, with descriptions parsed from the first sentence of a Javadoc comment. This dataset has been constructed from a much larger dataset of 51M java methods (Lopes et al., 2010) by filtering for English comments, removing pairs with less than three and more than 13 tokens in the description, or more than 100 tokens in the method. A target description contains 7.6 tokens on average. We reuse the provided training, validation and test split, which has been created by splitting per project. We report case-insensitive corpus-level BLEU scores (Papineni et al., 2002) and use the evaluation scripts released by LeClair and McMillan (2019) along with the dataset.

The `CodeSearchNet` dataset consists of function/documentation pairs in 6 different programming languages (Go, Java, JavaScript, PHP, Python, Ruby). The main task for this dataset is code search (i.e. given a documentation find the correct function), but it also has been applied to code-to-documentation generation by Feng et al. (2020) (CodeBERT). After pre-training CodeBERT on the full dataset, the authors filtered the dataset to remove samples with poor quality and subsequently fine-tune and evaluate the CodeBERT model for code-to-documentation generation on the filtered subset. To be comparable to CodeBERT we re-use only the filtered subset for training and evaluation of our model and use the same evaluation script[4] that computes smoothed BLEU-4 score as Feng et al. (2020). Furthermore we omit splitting on camel-case for this dataset, as this would yield a different tokenization that would influence the resulting BLUE score. We train a joint encoder on all languages, to leverage knowledge transfer to the less frequent languages.

Task 3: Neural Machine Translation To test our model on a non-code task, we evaluate it on machine translation on the IWSLT'14 English-

[4] Feng et al. (2020) use the same evaluation script for computing BLEU scores as Iyer et al. (2016).

Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Seq2Seq (Feng et al., 2020)†	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer (Feng et al., 2020)†	11.18	11.59	16.38	15.81	16.26	22.12	15.56
Roberta (Feng et al., 2020)†	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT (RTD+MLM) (Feng et al., 2020)†	12.16	14.90	18.07	**19.06**	17.65	**25.16**	17.83
Transformer (no tree) (Vaswani et al., 2017)	13.90	14.61	18.08	18.19	18.20	23.12	17.68
Relative Structural Transformer	**14.81**	**14.96**	**18.62**	17.93	**18.63**	23.77	**18.12**

Table 3: *Code captioning* on the `CodeSearchNet` dataset. As Feng et al. (2020) we report smoothed cumulative BLEU-4 scores. Results marked with † have been reported by Feng et al. (2020).

German (En-De and De-En) dataset. We replicate the training settings from Nguyen et al. (2020), thereby also using the same preprocessing script to create parse trees with the Stanford CoreNLP parser (Manning et al., 2014) and also report tokenized BLEU-4. We use the same form of BPE on terminal nodes as on the code-related tasks, but omit splitting on camel-case (10k subwords).

4.3 Hyperparameters and Setup

We implemented our model in PyTorch with the fairseq toolkit (Ott et al., 2019), on top of an existing transformer implementation. As our focus is on studying syntax-specific extensions, we refrain to a standard transformer architecture and optimize only hyperparameters relevant to our extensions: We replicate the same transformer architecture and most hyperparameters that have been used in the experiments of Nguyen et al. (2020), consisting of 6 transformer layers in the encoder and decoder, with 4 attention heads, 1024-dimensional feed-forward layers, $d = 512$ dimensional token embeddings and the sharing of the input and output embedding matrices in the transformer decoder. Like Nguyen et al. (2020), we train our model using the Adam optimizer and an inverse square root learning rate scheduler with a linear warm-up for $4,000$ updates up to a peak learning rate of $5\text{e}{-4}$ and a dropout rate of 0.3. For our final evaluations, we generate sequences with beam search using a beam width of 5 and additionally prohibit repeating n-grams of length 2.

We learn a Byte-Pair-Encoding of 16k sub-words on the training data for all code-related tasks. For the lowest common ancestor loss (Section 3.2) we sample $M{=}\min(n, 50)$ node pairs per method. We also optimize the following parameters manually on the validation set: the batch size, γ_{lca}, whether *path-length* or *movements* are used for relative position representations, the clamping value C (Section 3.1), and whether the model is trained with a joint vocabulary for encoder and decoder, in which case we share the encoder's token embedding matrix with the decoder. We conduct three runs with the best hyperparameter setting and report results for the model with the highest BLEU/F1 on the validation data. The full set of hyperparameters is provided in Appendix A.1.

For the transformer baseline we tokenize the source code in the same way as for ASTs (Section 4.1) and train a regular transformer model on the resulting token sequence (Vaswani et al., 2017) with the same set of hyperparameters.

5 Experimental Results

Tables 1 – 3 cover the three code-related tasks, comparing our approach with the state-of-the-art methods and a regular sequential transformer baseline. In Table 1 we additionally conduct experiments with a transformer that uses absolute tree positional embeddings (Shiv and Quirk, 2019). We investigate the use of our approach on natural language machine translation in Table 4. Finally, we also study the effect of our structural loss function and relative position representations in ablation studies.

Task 1: MethodNaming On the *method naming* task our model outperforms the state-of-the-art code2seq model (Alon et al., 2018a) on `java-small`, `java-med` and `java-large` datasets by more than 6.2%. We continuously outperform the absolute tree transformer (Shiv and Quirk, 2019), which is – on `java-small` – outperformed by a token-level transformer. On all three datasets our token-level transformer baseline outperforms the current SoTA (which does not use BPE), by up to 4.7%. This demonstrates the effectiveness of our preprocessing pipeline, particularly the use of BPE for source code. Our tree extensions improve performance further, especially on small

Model	IWSLT'14	
	En-De	De-En
Tree2Seq (Shi et al., 2018)†	24.01	29.95
Conv-Seq2Seq (Gehring et al., 2017)†	24.76	30.32
Transformer (Vaswani et al., 2017)†	28.35	34.42
Dynamic Conv (Wu et al., 2019)†	28.43	34.72
Hierarchical Transformer (Nguyen et al., 2020)†	**29.47**	**35.96**
Transformer (no tree) (Vaswani et al., 2017)	28.30	34.62
Relative Structural Transformer	29.40	35.32

Table 4: *Machine translation* on the `IWSLT'14` dataset. We report the tokenized cumulative BLEU-4 score. Results marked with † have been reported by Nguyen et al. (2020).

Relationship Type	C	γ_{lca}	P	R	F1
-	-	-	56.2	56.2	55.1
-	-	0.3	60.8	59.3	58.7
Movements	2	-	60.5	58.8	58.4
Movements	2	0.05	61.1	59.2	58.9
Movements	2	0.3	**61.3**	**60.0**	**59.4**
Path-Length	8	0.3	60.6	59.4	58.8

Table 5: Ablation study on `java-med`. All models are trained on linearized ASTs. Adding a structural loss or relative positions improves performance. Adding both gives best results.

datasets. We conclude that our approach successfully adds a structural prior. A t-SNE visualization in Figure 5 confirms that our extensions yield embeddings that are more closely aligned with the AST structure.

Task 2: CodeCaptioning On the *code-to-documentation* task on the `FunCom` dataset our model outperforms the state-of-the-art by 1.8%. Including structural information shows to be beneficial, as we outperform a token-level transformer baseline by 0.6%. On the `CodeSearchNet` dataset our approach – which is trained end-to-end – outperforms the language model based CodeBERT on most languages, with an overall improvement of 0.3%. Our baseline outperforms reported transformers by 2.1%, that have been using a non code-specific BPE[5] and no literal replacements.

Task 3: Machine Translation Our model is able to utilize the structural prior and improves over a token-level transformer baseline by up to 1.1%. It thereby outperforms various other recent models, and yields competitive results compared to the hierarchical transformer (Nguyen et al., 2020). Also, we found our approach to be much more scalable with respect to input sequence length (Figure 4 in the Appendix), both in terms of speed (left) and memory (right)[6]. Since sequence lengths and datasets in code-related tasks are longer than in machine translation, memory limitations prevent us from training the hierarchical transformer with adequate batch sizes. Figure 4 also shows that our approach's performance is comparable to that of a sequential transformer with only negligible speed and memory losses. Overall, this indicates that our method is also applicable to natural language in general and may be even used for longer documents.

Ablation Studies We investigate the impact of our two extensions on the `java-med` dataset. Table 5 shows that simply linearizing the AST without any structural information results in a performance drop of 4.3% compared to the best model. Using the structural loss or relative positions independently improves performance. Even though LCAs for ancestor prediction can be inferred from the relative positions, the system benefits from an additional implicit modeling as an auxiliary loss. We hypothesize that the two approaches complement each other, as LCA prediction enforces the model to represent relative position in the encodings $\mathbf{z}$.

6 Conclusion

We propose two extensions to transformer models with relative position representations to incorporate structural information from trees: (1) relative position representations that encode the position of nodes explicitly using a representation that encodes movements between any pair of nodes in the tree, and (2) a new structural loss based on lowest common ancestor (LCA) prediction. In a broader sense, our model offers a simple yet effective way to incorporate syntactic information into transformer models. This may be interesting for modeling long-term dependencies in NLP tasks such as relation extraction or co-reference resolution. Another interesting direction for future research would be to incorporate our structural extensions not only into supervised training, but also into language models, e.g. for an unsupervised learning of source code semantics.

[5] CodeBERT re-uses the BPE of RoBERTa.

[6] We used the reference implementation of the hierarchical transformer (`https://github.com/nxphi47/tree_transformer`) with the same hyperparameters as ours.

References

Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. 2019. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3159–3166.

Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2017. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182.

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

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

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018b. A general path-based representation for predicting program properties. In *ACM SIGPLAN Notices*, volume 53, pages 404–419. ACM.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40.

Eric O. D. Andriantiana, Kenneth Dadedzi, and Stephan Wagner. 2018. The ancestral matrix of a rooted tree.

Hlib Babii, Andrea Janes, and Romain Robbes. 2019. Modeling vocabulary for big code machine learning. *arXiv preprint arXiv:1904.01873*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Angela Fan, David Grangier, and Michael Auli. 2017. Controllable abstractive summarization. *arXiv preprint arXiv:1711.05217*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *CoRR*, abs/1811.01824.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. 2017. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

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

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code prediction by feeding trees to transformers.

Alex LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *2020 IEEE/ACM International Conference on Program Comprehension*.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. *CoRR*, abs/1902.01954.

Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *CoRR*, abs/1904.02660.

C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI source code data sets.

Senthil Mani, Anush Sankaran, and Rahul Aralikatte. 2019. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 171–179. ACM.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Xuan-Phi Nguyen, Shafiq Joty, Steven CH Hoi, and Richard Socher. 2020. Tree-structured attention with hierarchical accumulation. *arXiv preprint arXiv:2002.08046*.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

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

Haoyue Shi, Hao Zhou, Jiaze Chen, and Lei Li. 2018. On tree-based neural sentence modeling. *arXiv preprint arXiv:1808.09644*.

Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12081–12091. Curran Associates, Inc.

Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Hao Tian, Hua Wu, and Haifeng Wang. 2019. ERNIE 2.0: A continual pre-training framework for language understanding. *CoRR*, abs/1907.12412.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700.

Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. 2019. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018. Graph2seq: Graph to sequence learning with attention-based neural networks.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696.

A Appendices

A.1 Datasets and Hyperparameters

Relevant statistics about the datasets used are provided in Table 6 and additional hyperparameters for the experiments are listed in Table 7. Additionally, we provide code, data and configs for our experiments at `https://removed-for-blind-review.net`. We ran a hyperparameter sweep over the clamping distance and explored the values $C = \{2, 3, 8, 16\}$ for *movements* and $C = \{4, 6, 8, 16, 32\}$ for *path-length*. For γ_{lca} we investigated $\{0.05, 0.3, 0.6, 1\}$. We report the best performing hyperparameters in Table 7. For the absolute tree transformer (Shiv and Quirk, 2019) we ran a hyperparameter sweep over the maximum tree depth $k = \{32, 64, 256\}$.

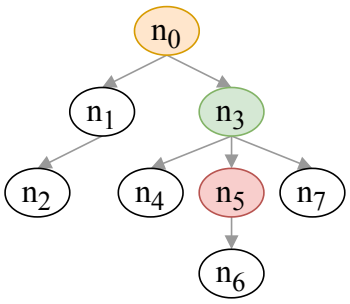
For the machine translation experiments on the `IWSLT'14` dataset we follow the approach of Nguyen et al. (2020) and use 5% of ≈160k sentence pairs for validation, thereby combine (IWSLT14.TED.dev2010, dev2012, tst2010-tst2012) for testing and also use Stanfords CoreNLP (v3.9.2) to parse trees. We average the 5 checkpoints with the best validation BLEU.

A.2 Infrastructure Details

We vary the amount of GPUs (all Nvidia GeForce GTX1080TI) by the amount of training data. We train models on `java-med`, `java-large` and code captioning datasets on 3 GPUs, but for `java-small` and the machine translation experiments in Table 4 we use a single GPU.

A.3 Sample Visualizations

To illustrate the effect of our structural loss on the transformer, Figure 5 visualizes the (t-SNE-transformed) representations $\mathbf{z}$ produced by the encoder. Edges in the visualization correspond to edges in the AST. When using only relative position representations, but no structural loss (right), AST nodes are scattered over the embedding space, where mostly the representations of identical input tokens form clusters (lower right). In contrast, our relative structural transformer using LCA-loss (left) produces clusters that strongly align with subtrees: Obviously, the representation of a code component (such as a for-loop) is similar to its subcomponents (such as the statements within the loop), an indicator that attention is aligned with the tree structure and focuses stronger on close components in the tree.



(a) Tree

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) Node incidence matrix $\mathbf{N}$ (Equation (5))

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 3 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 3 & 3 & 2 \\ 1 & 1 & 1 & 2 & 2 & 3 & 4 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 \end{bmatrix}$$

(c) Ancestral matrix $\mathbf{A}$ (Equation (6))

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 1 & 0 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 & 0 & 0 & 1 \\ 3 & 3 & 3 & 2 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(d) Movements matrix $\mathbf{M}$ (Equation (7))

Figure 2: Visualization of the matrices used for computing relative position representations for trees.

	java-small	java-med	java-large	CodeSearchNet	FunCom	IWSLT'14
Samples Train	665,115	3,004,536	15,344,512	908,224	1,937,136	160,239
Samples Valid	23,505	410,699	320,866	44,689	106,153	7,283
Samples Test	56,165	411,751	417,003	52,561	52,561	6,750

Table 6: Statistic about the datasets used.

	java-small		java-med		java-large		CodeSearchNet		FunCom		IWSLT'14	
	Baseline	RST	Baseline	RST	Baseline	RST	Baseline	RST	Baseline	RST	Baseline	RST
Warmup Updates	4000	4000	4000	4000	10000	10000	4000	4000	4000	4000	4000	4000
Max Epoch	30	30	60	60	60	60	60	60	60	60	70	70
Validation Metric	F1	F1	F1	F1	F1	F1	BLEU	BLEU	BLEU	BLEU	BLEU	BLEU
Validation Performance	44.84	46.19	56.05	58.61	63.31	63.66	8.32*	8.92*	23.42	23.91	29.69	30.07
Max Source Positions	512	512	512	512	512	512	1024	1024	512	512	1024	1024
Max Target Positions	80	80	80	80	80	80	1024	1024	30	30	1024	1024
Batch Size (in tokens/batch/gpu)	8192	8192	8192	8192	8192	8192	6146	6146	6146	6146	4096	10240
Accumulate Gradients (in batches)	8	8	8	8	8	8	6	6	6	6	-	-
Share Embeddings (between Encoder/Decoder)	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No	No
Relationship Type	-	Movements	-	Movements	-	Path-Length	-	Movements	-	Movements	-	Path-Length
C	-	2	-	2	-	8	-	2	-	2	-	8
γ_{lca}	-	0.3	-	0.3	-	0.3	-	0.3	-	0.3	-	0.3
Parameters (Million)	38.76	38.79	39.3	39.9	47.5	47.6	39.8	40.4	47.5	48.3	39.5	40
Average Runtime (hours)	10	12	22	25	66	71	20	23	16	19	7	9

Table 7: Additional hyperparameters for the experiments in Table 1 – 4. (*) denotes that we used a different BLEU implementation during validation than for testing. For all experiments we set *Label Smoothing*=0.1, *Learning Rate*=5e-4, *Optimizer*: Adam, *Adam-Betas*=0.9, 0.98 and *Weight Decay*=0.0001

```
boolean isPrime(int x) {
    double mRoot = Math.sqrt(x);
    for(int i=2; i <= mRoot; ++i)
    {
      if(x % i == 0){
        return false;
      }
    }
    return true;
}
```

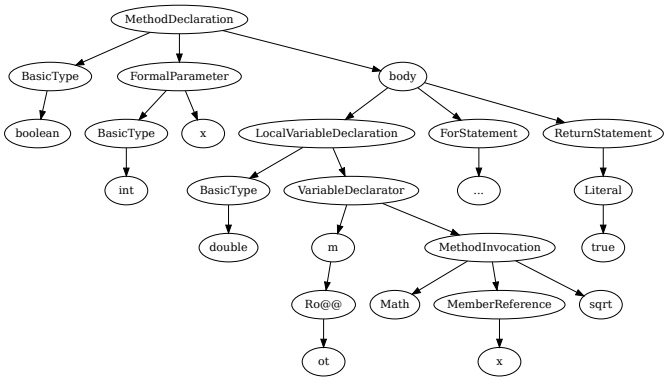


Figure 3: A java method that computes whether a number is a prime, together with its preprocessed abstract syntax tree.

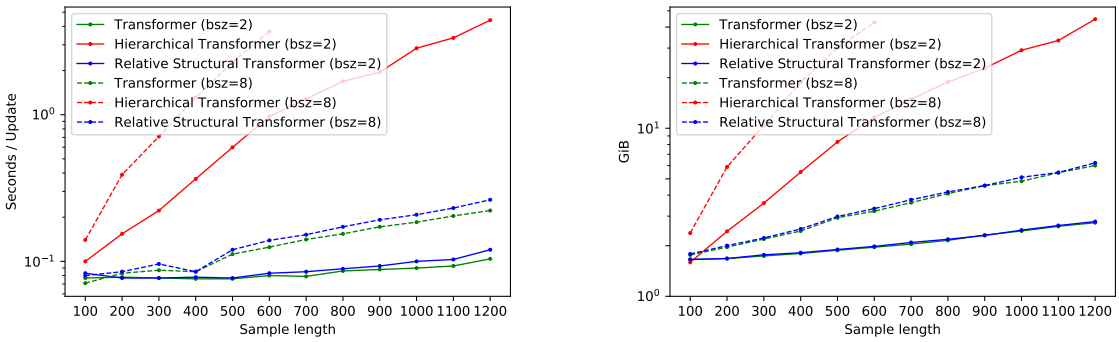


Figure 4: Memory usage (GiB) and speed comparison (seconds per iteration) with respect to sequence length using a fixed batch-size (bzs – in samples per batch). We compare a regular transformer – whose results are also applicable to the absolute tree transformer (Shiv and Quirk, 2019), as absolute positional representations can be pre-computed, the hierarchical transformer (Nguyen et al., 2020) using the released reference implementation and our relative structural transformer utilizing all proposed extensions. All benchmarks are done with the same model architecture (in terms of layers, dimensions and vocabulary), the same datasets – containing only samples of a specific length – on a single Quadro RTX 8000 (48GiB).

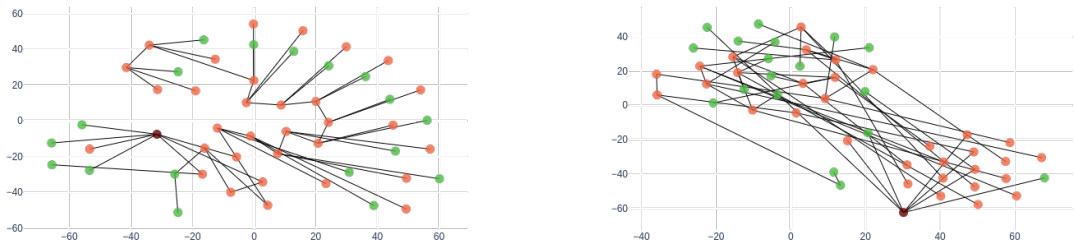


Figure 5: t-SNE visualization of the encoded node representations of a model trained with (left) and without (right) a structural loss on `java-med`. The root node is marked dark red, artificial/abstract tokens – that don't appear in source code – are red and nodes that appear in source code green. Nodes are connected by lines, as in the original AST.

```
public Throwable blockingGetError() {
    if (getCount() != 0) {
        try {
            BlockingHelper.verifyNonBlocking();
            await();
        } catch(InterruptedException ex) {
            dispose();
            return ex;
        }
    }
    return error;
}
```

Target: ”Block until the latch is counted down and return the error received or null if no error happened .”
Transformer (no tree): ”Returns an error if there is one . Otherwise returns nil .”
Relative Structural Transformer: ”This method blocks until there is an error or the end of the queue is reached .”

```
public static ScheduledExecutorService create(ThreadFactory factory) {
    final ScheduledExecutorService exec = Executors.newScheduledThreadPool(1, factory);
    tryPutIntoPool(PURGE_ENABLED, exec);
    return exec;
}
```

Target: ”Creates a ScheduledExecutorService with the given factory .”
Transformer (no tree): ”Create a new ScheduledExecutorService .”
Relative Structural Transformer: ”Creates a ScheduledExecutorService with the given ThreadFactory .”

Figure 6: Sample predictions on `CodeSearchNet`.