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

Abstract

We suggest two extensions to incorporate syntactic information into transformer models operating on linearized trees (e.g. abstract syntax trees). 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, and demonstrate how those movements can be computed efficiently on the fly. Second, we train the network to predict the lowest common ancestor of node pairs using a new structural loss function. 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 to 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). Thereby, 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 a transformer operates on the resulting node sequence. This transformer is extended to take structure into account, using 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. 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. Additional loss functions – such as next sentence prediction (Devlin et al., 2018) or sentence reordering (Sun et al., 2019) – have already been shown to improve the accuracy of transformer networks. Following this line of thought, we suggest a new loss function based on the prediction of lowest common ancestors.

3. We explore the two above strategies in experiments with a focus on source code, and demonstrate substantial improvements on all five datasets tested. Combining both methods yields the best results, outperforming the state-of-the-art on method naming by 6% and offering significant improvements over sequential transformer baselines.

2 Approach

As illustrated in Figure 1, our model 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 dependency or constituency parse trees. Specifically, we present two extensions: First, we replace absolute positional embeddings with self-attention with relative position representations (Shaw et al., 2018) and thereby encode the pairwise relations between nodes as movements between them (Section 2.2). Second, we introduce a structural loss enforcing the model to predict the lowest common ancestor of two nodes (Section 2.3) based on the encoder output.

We feed the tree into the transformer by replacing the conventional sequence of tokens with a sequence of nodes (including both terminals and non-terminals) using a pre-order traversal of the tree. This yields a sequence of nodes \( x = (x_1, x_2, x_3, \ldots, x_n) \) (where \( x_1 \) is the root node). 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 \( i \) and \( u \) (or \( \text{depth}(i) \)) denotes the depth of node \( i \). Based on this path, we define the ancestors and descendants of Node \( i \):

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

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:

\[
\text{lca}(i,j) = \arg\max_{a \in \text{anc}(i) \cap \text{anc}(j)} \text{depth}(a)
\]

### 2.1 Model Architecture

Our model uses a regular transformer architecture which has become ubiquitous in sequence-to-sequence tasks such as neural machine translation (Vaswani et al., 2017) for brevity we omit details of the transformer architecture. The node sequence \( x \) is first transformed into a sequence of real-valued input embeddings with \( x_i \in \mathbb{R}^d \). These are processed by the transformer encoder, resulting in a sequence of continuous representations \( z(x) \), or shorter \( z = (z_1, \ldots, z_n) \). From this, the auto-regressive transformer decoder generates an output token sequence \( y = (y_1, \ldots, y_m) \), in our case a summary of the input program \( x \). When generating token \( y_{i+1} \), the decoder attends to the whole encoded sequence \( z \) as well as all previously generated symbols \( y_1, \ldots, y_i \).

#### 2.2 Self-Attention with Relative Position Representations

Shaw et al. (2018) replace absolute positional embeddings in transformers with relative position representations (RPR). Those 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 \( x^L = (x^L_1, \ldots, x^L_n) \) with \( x^L_i \in \mathbb{R}^d \) and outputs a new sequence \( x^{L+1} = (x^{L+1}_1, \ldots, x^{L+1}_n) \in \mathbb{R}^{d_h} \). The pairwise relationship between input elements \( x^L_i \) and \( x^L_j \) is represented by learned embeddings \( a_{ij} \in \mathbb{R}^{d_h} \) that are shared between attention heads. The output of self-attention with RPR is computed by

\[
x^{L+1}_i = \sum_{j=1}^{n} \alpha_{ij} (x^L_j W^V) \tag{1}
\]
whereas the weight coefficient $\alpha_{ij}$ is computed by a softmax over compatibility scores $e_{ij}$:

$$ e_{ij} = \frac{x^T_i W^Q (x^T_j W^K + a_{ij})^\top}{\sqrt{d}} $$

where $d \times d_h$ matrices $W^Q$, $W^K$, $W^V$ map the inputs to a head-specific embedding space.

Note that Equation (2) is the original self-attention from Vaswani et al. (2017) if one omits $a_{ij}$. Shaw et al. (2018) define the relative positional embeddings $a_{ij}$ based on the linear distance between sequential input tokens, clamped to a minimum and maximum distance.

We extend relative position representations $a_{ij}$ to take the input tree structure into account, based on the relative position of nodes $i$ and $j$. A matrix $M \in \mathbb{R}^{n \times n}$ encodes the paths between nodes: From $i$, we first take $M_{ij}$ steps upward to $\text{lca}(i,j)$ and then $M_{ji}$ steps downward to $j$. We also take into account whether a node $i$ is left of node $j$ ($1_{i<j}$) and investigate two options for deriving relative position representations from this path:

- **Path length**: We use the path length (clamped to a maximum of $C$) $l(i,j) = M_{ij} + M_{ji} \in \{0, ..., C\}$ and derive the relative position representations from an embedding table $E$:

$$ a_{ij} := E_{1_{i<j}; l(i,j)} $$

- **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 $x_1$ ($-2[1]1$). Both values are used as indices to the embedding table:

$$ a_{ij} := E_{1_{i<j}; M_{ij}, M_{ji}} $$

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

**Computing relative node positions efficiently**

The basis for relative position representations is the matrix $M$. We demonstrate that $M$ – and with it $a_{ij}$ – can be derived efficiently with matrix operations on the fly.

We first represent the tree using a binary node incidence matrix $N \in \{0, 1\}^{n \times n}$ (see Appendix A for a visualization) which encodes each node’s path to the root by

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

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

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

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

$$ A = NN^\top $$

- the movement matrix $M \in \mathbb{R}^{n \times n}$ by

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

Note that we can easily derive $N$ from the size of each node’s sub-tree, by filling $[\text{desc}(j)]+1$ rows in column $j$ with ones, starting at the diagonal ($N_{ij}=1$ for $i = j, j+1, ..., j+|\text{desc}(j)|$).

The number of descendants per node can be pre-computed in $O(n)$ total, 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 sufficient for input lengths commonly used in NLP (e.g., 1024).

**2.3 Structural Loss**

The goal of our structural loss is to enforce the encoded representations $z$ 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 code primitive (e.g., an if-statement), while nodes representing different passages in a program are considered as “dissimilar”. This notion of similarity can be enforced by training our model to predict node pairs’ LCAs. To do so, we concatenate the encoded representations $z_i, z_j$ of two nodes to predict their lowest common ancestor $a$:

$$ v_{ij} = \text{RELU}([z_i; z_j] \cdot A + b) $$

$$ p^{\text{ln}}(a|i,j) = \text{softmax}(v_{ij} : Z)_a $$
where $A \in \mathbb{R}^{2d \times d}$ and $b \in \mathbb{R}^d$ are learned parameters, and $Z \in \mathbb{R}^{d \times n}$ 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.

**Loss Function** For lowest common ancestor prediction we define a log-likelihood loss over node pairs $(i, j)$:

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

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

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

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

**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 proportional 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 $a+1, \ldots, a+|\text{desc}(a)|$, and we sample uniformly from this range. More precisely, we distinguish two cases: (1) If $a$ has only a single child, we choose $i=a$ and $j$ as a random descendant of $a$. Note that when $i$ and $j$ were two descendants of $a$, their LCA would not be $a$ but $a$’s child or one of its descendants. (2) If $a$ has at least two children, we draw samples $i, j$ from two different children (or their descendants).

### 3 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. Additionally, we test our model on neural machine translation on natural language.

#### 3.1 Preprocessing Source Code

When parsing code to ASTs, 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 (Senrich et al., 2015). The example from before might become “get Num@@ber Hex 16”.
3. To reduce the vocabulary even 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 digits with only 20 tokens in our vocabulary. The example becomes “get Num@@ber Hex 100 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

---

1 We use the tree-sitter library for parsing code into ASTs.
token \( t_i \) with \( x_i \)'s parent as parent and each other token with its predecessor as parent. In the example, we obtain get←Num@←bert←Hex←1@←6. Note that the BPE encoding is removed when computing evaluation measures such as BLEU for reasons of comparability with prior work.

3.2 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 3.1). 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 introduced 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).

After pre-training CodeBERT on the full dataset, Feng et al. (2020) 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\(^*\) 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 joined 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-German (En-De) 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). We use the same form of Byte-Pair Encoding (10k subwords) on terminal nodes as on the code-related tasks, but omit splitting on camel-case. Like Nguyen et al. (2020), we report tokenized BLEU-4.

3.3 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

\( ^* \)Feng et al. (2020) use the same evaluation script for computing BLEU scores as Iyer et al. (2016).
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. Just as 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 \( 5 \times 10^{-4} \) 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-Encodings of 16k subwords on the training data for all code-related tasks. For the lowest common ancestor loss (Section 2.3) we sample \( M = \text{min}(n, 50) \) node pairs per method. We also optimize the following parameters manually on the validation set: the batch size, \( \gamma \text{leak} \), whether \textit{path-length} or \textit{movements} are used for relative position representations, and the clamping value \( C \) (Section 2.2) 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 3.1, Steps 1.-4.), and train a regular transformer model on the resulting token sequence (Vaswani et al., 2017) with the same set of hyperparameters.

Table 1: Comparison with SoTA on the \textit{method naming} task. (*) denotes that the authors report macro F1, whereas all other results are micro F1. Results marked with † are taken from Alon et al. (2018a).

<table>
<thead>
<tr>
<th>Model</th>
<th>java-small</th>
<th>java-med</th>
<th>java-large</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F1</td>
</tr>
<tr>
<td>Paths+CRFs (Alon et al., 2018b)†</td>
<td>8.4</td>
<td>5.6</td>
<td>6.7</td>
</tr>
<tr>
<td>code2vec (Alon et al., 2019)†</td>
<td>18.5</td>
<td>18.7</td>
<td>18.6</td>
</tr>
<tr>
<td>TreeLSTM (Tai et al., 2015)†</td>
<td>40.0</td>
<td>31.8</td>
<td>35.5</td>
</tr>
<tr>
<td>2-layer BiLSTM†</td>
<td>42.6</td>
<td>30.0</td>
<td>35.2</td>
</tr>
<tr>
<td>StructSum (Fernandes et al., 2018)*</td>
<td>-</td>
<td>-</td>
<td>61.4</td>
</tr>
<tr>
<td>ConvAttention (Allamanis et al., 2016)†</td>
<td>50.3</td>
<td>24.6</td>
<td>33.1</td>
</tr>
<tr>
<td>Transformer (no tree) (Alon et al., 2018a)†</td>
<td>38.1</td>
<td>26.7</td>
<td>31.4</td>
</tr>
<tr>
<td>code2seq (Alon et al., 2018a)†</td>
<td>50.6</td>
<td>37.4</td>
<td>43.0</td>
</tr>
<tr>
<td>Transformer (no tree) (Vaswani et al., 2017)</td>
<td>48.5</td>
<td>45.9</td>
<td>45.9</td>
</tr>
<tr>
<td>Relative Structural Transformer</td>
<td>52.7</td>
<td>47.6</td>
<td>48.6</td>
</tr>
</tbody>
</table>

4 Experimental Results

Tables 1 – 3 cover the three code-related tasks, comparing our results with the state-of-the-art and with a regular token-level transformer baseline. It is worth noting that throughout all code tasks, this baseline already performs competitive and outperforms reported transformer baselines without BPE by a large margin. This demonstrates the effectiveness of our preprocessing pipeline, particularly the use of BPE for source code. Finally, we also study the effect of our structural loss function and relative position representations using ablation studies.

Task 1: MethodNaming On the \textit{method naming} task our model outperforms the state-of-the-art on \textit{java-med} and \textit{java-large} datasets by more than 6.2%. On \textit{java-small}, we achieve better results than code2seq but are outperformed by the structured summarization model (Fernandes et al., 2018). However, this approach has been reported to be outperformed on \textit{java-large} by code2seq (Alon et al., 2018a). Note that Fernandes et al. (2018) report macro F1, while code2seq uses micro F1. On all three datasets our token-level transformer baseline outperforms the current SoTA (which does not use BPE), by up to 4.7%. Our tree extensions improve performance further, especially on small datasets. We conclude that our approach successfully adds a structural prior.

Task 2: CodeCaptioning On the \textit{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.
Table 2: Comparison with SoTA on code captioning on the FunCom dataset (java). We report the cumulative BLEU-4 score, together with the 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) and ‡ by Alex LeClair (2020).

<table>
<thead>
<tr>
<th>Model</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>BLEU-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer (no tree)</td>
<td>40.3</td>
<td>23.6</td>
<td>16.4</td>
<td>12.6</td>
<td>21.1</td>
</tr>
<tr>
<td>Relative Structural Transformer</td>
<td>42.3</td>
<td>24.4</td>
<td>16.8</td>
<td>12.9</td>
<td>21.7</td>
</tr>
</tbody>
</table>

Table 3: Comparison with SoTA on 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).

<table>
<thead>
<tr>
<th>Model</th>
<th>Ruby</th>
<th>Javascript</th>
<th>Go</th>
<th>Python</th>
<th>Java</th>
<th>Php</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq (Feng et al., 2020)†</td>
<td>9.64</td>
<td>10.21</td>
<td>13.98</td>
<td>15.93</td>
<td>15.09</td>
<td>21.08</td>
<td>14.32</td>
</tr>
<tr>
<td>Transformer (Feng et al., 2020)†</td>
<td>11.18</td>
<td>11.59</td>
<td>16.38</td>
<td>15.81</td>
<td>16.26</td>
<td>22.12</td>
<td>15.56</td>
</tr>
<tr>
<td>Roberta (Feng et al., 2020)†</td>
<td>11.17</td>
<td>11.90</td>
<td>17.72</td>
<td>18.14</td>
<td>16.47</td>
<td>24.02</td>
<td>16.57</td>
</tr>
<tr>
<td>CodeBERT (RTD+MLM) (Feng et al., 2020)†</td>
<td>12.16</td>
<td>14.90</td>
<td>18.07</td>
<td>19.06</td>
<td>17.65</td>
<td>25.16</td>
<td>17.83</td>
</tr>
<tr>
<td>Transformer (no tree) (Vaswani et al., 2017)</td>
<td>13.90</td>
<td>14.61</td>
<td>18.08</td>
<td>18.19</td>
<td>18.20</td>
<td>23.12</td>
<td>17.68</td>
</tr>
</tbody>
</table>

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 a token-level transformer baseline by 1.1%. It performs slightly worse than the hierarchical transformer (Nguyen et al., 2020), while still competitive. This shows that our method applicable to seq2seq problems on natural language in general.

**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 one might argue that the ancestor prediction can be inferred from the relative positions, a combination of both yields the highest performance. A t-SNE visualization in Figure 4 indicates that our extensions yield embeddings that are more closely aligned with the AST structure.

5 Related Work

Abstractive summarization condenses a source sequence into a short descriptive target sequence, while maintaining its meaning (Fan et al., 2017), and has thus 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, to reduce the out-of-vocabulary problem by splitting words into a fixed set of n-grams (Sennrich et al., 2015). Modeling the semantics of source code by pre-

$^5$CodeBERT re-uses the BPE of RoBERTa.
dicting 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) have merged from NLP into modeling semantics of source code, 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 predicts the lowest common ancestor between node pairs, what is related to encoding the path between two nodes. Fernandes et al. (2018) and Alex LeClair (2020) propose a structured summarization approach for

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree2Seq (Shi et al., 2018)</td>
<td>24.01</td>
</tr>
<tr>
<td>Conv-Seq2Seq (Gehring et al., 2017)</td>
<td>24.76</td>
</tr>
<tr>
<td>Transformer (Vaswani et al., 2017)</td>
<td>28.35</td>
</tr>
<tr>
<td>Dynamic Conv (Wu et al., 2019)</td>
<td>28.43</td>
</tr>
<tr>
<td>Hierarchical Transformer (Nguyen et al., 2020)</td>
<td>29.47</td>
</tr>
<tr>
<td>Transformer (no tree) (Vaswani et al., 2017)</td>
<td>28.30</td>
</tr>
<tr>
<td>Relative Structural Transformer</td>
<td>29.40</td>
</tr>
</tbody>
</table>

Table 4: Comparison with SoTA on machine translation on the IWSLT’14 (En-De) dataset. We report the tokenized cumulative BLEU-4 score. Results marked with † have been reported by Nguyen et al. (2020).

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>C</th>
<th>γlca</th>
<th>P</th>
<th>R</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movements</td>
<td>2</td>
<td>0.05</td>
<td>61.1</td>
<td>59.2</td>
<td>58.9</td>
</tr>
<tr>
<td>Movements</td>
<td>2</td>
<td>0.3</td>
<td>61.3</td>
<td>60.0</td>
<td>59.4</td>
</tr>
<tr>
<td>Path-Length</td>
<td>8</td>
<td>0.3</td>
<td>60.6</td>
<td>59.4</td>
<td>58.8</td>
</tr>
</tbody>
</table>

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

6 Conclusion

We have proposed two extensions to transformer sequence-to-sequence models to incorporate information from syntax trees: (1) relative position representations that encode the position of nodes explicitly, and (2) a new structural loss based on lowest common ancestor (LCA) prediction. Even though the LCA can be derived from relative position representation, our ablation study showed that a combination of both methods to be most effective. We hypothesize that the two approaches complement each other, as LCA prediction enforces the model to represent relative position in the encodings z. 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 research direction would be the combination of language models and our structural transformer, e.g. for modeling the semantics of source code.
References


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


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


A Appendices

A.1 Datasets and Hyperparameters

Relevant statistics about the datasets used are provided in 6 and additional hyperparameters that are used 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 $\gamma_{lca}$ we investigated $\{0.05, 0.3, 0.6, 1\}$. We report the best performing hyperparameters in Table 7.

For the machine translation experiments on the IWSLT’14 dataset we follow the approach of Nguyen et al. (2020) and use 5% of $\approx 160k$ 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 the experiments on java-small and the machine translation experiments in Table 4 we use only a single GPU.

A.3 Sample Visualizations

To illustrate the effect of our structural loss on the transformer, Figure 4 visualizes the (t-SNE-transformed) representations $z$ 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.
### Table 6: Statistic about the datasets used.

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

### 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

<table>
<thead>
<tr>
<th></th>
<th>java-small</th>
<th>java-med</th>
<th>java-large</th>
<th>CodeSearchNet</th>
<th>FunCom</th>
<th>IWSLT’14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warmup Epoch</td>
<td>4000</td>
<td>4000</td>
<td>4000</td>
<td>10000</td>
<td>10000</td>
<td>4000</td>
</tr>
<tr>
<td>Max Epoch</td>
<td>30</td>
<td>30</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Validation Metric</td>
<td>F1</td>
<td>F1</td>
<td>F1</td>
<td>F1</td>
<td>F1</td>
<td>BLEDU</td>
</tr>
<tr>
<td>Validation Performance</td>
<td>44.84</td>
<td>46.19</td>
<td>56.05</td>
<td>63.61</td>
<td>63.66</td>
<td>8.32</td>
</tr>
<tr>
<td>Max Source Positions</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Max Target Positions</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>1024</td>
</tr>
<tr>
<td>Batch Size (in tokens/batch/gpu)</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
<td>6146</td>
</tr>
<tr>
<td>Accumulate Gradients (in batches)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Share Embeddings (between Encoder/Decoder)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Relationship Type</td>
<td>- Movements - Movements - Path-Length - Movements - Movements - Path-Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>- 2</td>
<td>- 2</td>
<td>- 2</td>
<td>- 2</td>
<td>- 2</td>
<td>- 2</td>
</tr>
<tr>
<td>γlca</td>
<td>- 0.3</td>
<td>- 0.3</td>
<td>- 0.3</td>
<td>- 0.3</td>
<td>- 0.3</td>
<td>- 0.3</td>
</tr>
<tr>
<td>Parameters (Million)</td>
<td>38.76</td>
<td>38.79</td>
<td>39.3</td>
<td>39.9</td>
<td>47.5</td>
<td>47.6</td>
</tr>
</tbody>
</table>

Average Runtime (hours) 10 12 22 25 66 71 20 23 16 19 7 9

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

Figure 4: 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.
```java
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."

```java
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 5: Sample predictions on CodeSearchNet.