NOVEL POSITIONAL ENCODINGS TO ENABLE TREE-STRUCTURED TRANSFORMERS

Abstract

With interest in program synthesis and similarly flavored problems rapidly increasing, neural models optimized for tree-domain problems are of great value. In the sequence domain, transformers can learn relationships across arbitrary pairs of positions with less bias than recurrent models. Under the intuition that a similar property would be beneficial in the tree domain, we propose a method to extend transformers to tree-structured inputs and/or outputs. Our approach abstracts transformer’s default sinusoidal positional encodings, allowing us to substitute in a novel custom positional encoding scheme that represents node positions within a tree. We evaluated our model in tree-to-tree program translation settings, achieving superior performance to other models from the literature on several tasks.

1 Introduction

Neural networks have been successfully applied to an increasing range of tasks, including speech recognition, machine translation and image recognition. These crucially depend on techniques for representing one dimensional audio and text streams as well as two dimensional images. For decades now, researchers have relied on convolutional and recurrent techniques for processing inputs with large or dynamic sizes. However, recent efforts in sequence processing using attention based models Vaswani et al. (2017) have produced new state-of-the-art results in difficult tasks such as machine translation Wu et al. (2016). These techniques allow information to flow over unbounded distances during training and inference, without the need for complex gates and gradient clipping. This type of long distance flow, driven by learned attention transforms over positional encoding, provides a powerful computational mechanism.

Another recent line of work has begun to apply neural networks to programming tasks Allamanis et al. (2018). In recent years, programming language analysis techniques have begun to exploit statistical techniques commonly used on large natural language corpora Hindle et al. (2012). These can be used to identify idioms in software, enable searching for code clones, searching code by natural language, or even translating from one programming language to another. However, representing programs is an interesting challenge.

We could see them as a one dimensional sequence of tokens and use techniques common in the natural language programming literature. Yet these programs are intentionally endowed with hierarchical structure, even graph-like relations. A common approach is to pass information through neighbors in the graph, in a manner that is reminiscent of message passing in graphical models Li et al. (2016). To ensure that information can fully propagate across the graph, this message-passing must be applied multiple times, bounded by the diameter of the graph.

We introduce novel positional encodings of tree-structured data. Using this encoding, we can apply transformers to any number of tree structured settings, ranging from natural language parse trees to program abstract syntax trees. This allows information to percolate fully across the graph in a single layer. We evaluate these encodings on the task of programming language translation (e.g., translating JavaScript to CoffeeScript) Chen et al. (2018), setting a new state-of-the-art in several settings.
1.1 Transformers: Non-recurrent Sequence Modeling

Recurrent architectures that encode sequences into fixed-length representations have some disadvantages. From a generalization perspective, recurrent cells face the challenge of learning relationships between tokens many time steps apart. From an efficiency standpoint, recurrence does not lend itself to parallelism, often rendering recurrent models expensive to train. The transformer [Vaswani et al., 2017] is a stateless sequence-to-sequence architecture motivated by these issues, designed by extensively using attention mechanisms in lieu of state. Its design has the additional advantage of easier interpretation; its attention layers can at least reveal information about learned relationships between elements of a sequence. LSTM-style models, in contrast, employ an obtuse series of neural layers between timesteps that render such relationships unclear. We would ideally like to capture the benefits of the transformer’s stateless architecture when modeling trees, and potentially other modalities beyond sequences.

1.2 Stateless Tree Modeling

In this work, we generalize transformers to embed tree representations. Based on a novel scheme of custom positional encodings we construct for tree structures, we propose tree-transformers for non-recurrent tree-to-tree learning. We present results on the performance of topiary networks on program translation benchmarks.

2 Bag-of-Positions Interpretation

The order of a sequence is rich in information, and order-agnostic (bag-of-words) models are limited in power by their inability to use this information. The most common way to capture order is through recurrence; recurrent models inherently consider the order of an input sequence by processing its elements sequentially. In the absence of recurrence, we require additional information about the input sequence’s order in some other form. Transformer models provide this additional information in the form of positional encodings. Each position in the input sequence is associated with a vector, which is added to the embedding of the token at that position. This allows the transformer to learn positional relationships between tokens as well as relationships between token embedding and positional encoding space.

While adding in positional encodings addresses the power limitations of bag-of-words representations, it fundamentally does so by upgrading the bag of words to a bag of annotated words. Indeed, the transformer’s core attention mechanism is order-agnostic, treating keys as a bag. The calculations performed on any given element of a sequence are entirely independent of the order of the rest of that sequence in that layer; this leaves most of the work of exploiting positional information to the positional encodings and autoregressive property.

Now, a bag of words annotated with positions can be equivalently thought of as a bag of positions annotated with words. From this perspective, we see that is is not at all necessary that our input “sequence” of positions have any direct correspondence with the sequence of associated “indices,” i.e. an evenly distributed number line. The original transformer’s positional encodings do form this correspondence for the purposes of sequence modeling, but we can consider more arbitrary positional encodings to represent more arbitrary structures within positional space, as long as the relationships between points in positional space have some useful semantic meaning. In particular, we can use this idea to extend the transformer to the tree domain.

3 Tree Positional Encodings

Now we construct our positional encoding scheme for trees. The transformer’s original positional encodings has two key properties we would ideally like to preserve. First, every position has a unique positional encoding, so that attention to any given position can be sharply defined. Second, any relationship between two positions can be modeled by an affine transform between their positional encodings. This allows the transformer to efficiently learn relationships between positions within its embedding layers. In the context of sequences, the relationship between two positions is simply the distance \( k \) that separates them. For trees though, the relation between two nodes is a path: a series
of steps up or down along tree branches. Our positional encoding scheme should try to associate each such path with a unique linear transform in a consistent way.

From a given node in an \( n \)-ary tree, there are \( (n + 1) \) potential length-1 paths: a branch down to each of \( n \) children, and a branch up to the parent. We will denote the branches down to children as \( D_1, \ldots, D_n \) and the branch up to parent as \( U \). We can then denote any longer path as a composition of these \( D \)'s and \( U \)'s, which act as operators. For example, if we wish to denote a node \( x \)'s grandparent's second child, we can write \( D_2 U U x \). Every path can be broken down into a composition of these \( (n + 1) \) operators, so we need only focus on their relationships. We want to associate each operator \( D_i \) with a unique affine transform; for convenience, we will simply refer to their affine transforms as \( D_i \) as well, and to \( x \)'s positional encoding as simply \( x \).

The fundamental relationship between these operators is that traveling up a branch negates traveling down any branch. Our constraint then is:

\[
UD_i = I \forall i \in \mathbb{N}^n
\]

The positional encoding scheme we propose adheres to this constraint for all trees up to a specified depth, and still works well in practice for even deeper trees. We will first explain the parameter-free form of our positional encoding scheme for simplicity. Our scheme takes two hyperparameters: \( n \), the degree of our tree, assumed to be regular; and \( k \), the maximum tree depth for which our constraint is preserved. Each positional encoding has dimension \( n \cdot k \). We assign the root a zero vector, and define every other node by its path from the root vector:

\[
x = D_{b_L}D_{b_{L-1}} \cdots D_10
\]

where \( b \) is a sequence of branch choices and \( L \) is the layer at which \( x \) resides.

Now we describe the affine transform of \( D_i \). We represent a move down along \( x \)'s \( i \)th branch by concatenating a one-hot \( n \)-vector with hot bit \( i \) to the left side of \( x \), and truncating \( x \) on the right to preserve dimensionality. We define \( U \) complementarily. In other words,

\[
D_i x = e^{p_i}_n; x[: -n] \\
U x = x[n : ]; 0_n
\]

These \( D, U \) satisfy our constraint whenever \( L \leq k \). Note that for trees with depth greater than \( k \), \( UD_i \) is not necessarily the identity. Traveling down more than \( k \) layers will cause this scheme to “forget” nodes more than \( k \) layers up, which cannot be inverted.

The parameter-free positional encoding scheme as proposed so far, while fulfilling the uniqueness property and approximately the linear composition property, lacks richness. It is analogous to a simplified sequential positional encoding scheme that simply defines the positional encoding at index \( i \) to be the number \( i \). The transformer instead opted for a rich stack of sinusoids of different frequencies to attend to a much wider variety of relationships. In a similar vein, we propose adding a parametrizable component to diversify our encodings.

Our encoding consists of a sequence of one-hot chunks, each representing a different layer of the tree. One will note that we can weigh these one-hot chunks with any geometric series without disrupting the affine property:

\[
x' = x \odot (1_n; p_n; p^2_n; \ldots)
\]

\( x' \) here meets the same properties as \( x \). Here, \( p \) is a parameter and \( p_n \) is a \( n \)-vector of \( p \)'s. As Figure 1 shows, different values for \( p \) result in radically different attention biases. Analogous to the stack of sinusoidal encodings, we propose a stacking multiple tree encodings, each equipped with its own \( p \) to be learned. To prevent the encodings’ norms from exploding, we apply \( tanh \) to \( p \) to bound it between -1 and 1, and normalize the encodings by a factor of \( \sqrt{1 - p_2} \) to bound the norm below 1.

4 Decoder

To accommodate a new positional encoding scheme, we need to slightly modify the decoder. The original transformer’s decoder concatenates a start token to the beginning of the sequence without modifying the positional encodings. This results in misalignment between autoregressed outputs
Figure 1: Nearest neighbor heatmaps of tree encodings with various values of $p$. We number the nodes in the tree according to a breadth-first left-to-right traversal of a balanced binary tree: position 0 is the root, 1 is the first child of root, 2 is the second child of root, 3 is the first child of the first child of root, and so on. In each case, we consider the row position as a “query” and each column position as a potential “value”. The attention score of solely the positional encoding after softmax is represented as a heatmap scaling from black (0.0) through red and yellow to white (1.0).

In the absence of decay, many of the lower-level positions in the tree are quite similar. For example, position 5 (Root, $D_2$, $D_1$) is most similar to itself (score of 0.44), but quite similar to position 6 (Root, $D_2$, $D_2$) and position 3 (Root, $D_1$, $D_1$) with scores of 0.16. An appropriate level of decay allows each position to be uniquely identified as in (b); too much decay provides little additional information as in (c).

Figure 2: Common traversals and mixtures thereof can be represented as linear transforms. Using the position encoding described in this paper, finding the parent, left child, or right child of a given node can be represented as linear transforms $U$, $D_1$, and $D_2$. Complex traversals can be represented also as linear transforms by composing these operations. The attention heatmaps below demonstrate the similarity of tree positional encodings applied to different points in the tree when the “query” has been transformed before dot product with the value.
Table 1: Program accuracy data for synthetic tasks. The tree-transformer has clear advantages over the sequence-transformer for larger tasks, indicating that the custom positional encodings may be providing useful structural information.

<table>
<thead>
<tr>
<th>Model Description</th>
<th>SYN-S</th>
<th>SYN-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree-transformer, depth-first search</td>
<td>99.95</td>
<td>98.36</td>
</tr>
<tr>
<td>Tree-transformer, breadth-first search</td>
<td>99.76</td>
<td>97.56</td>
</tr>
<tr>
<td>Seq-transformer, parse trees</td>
<td>99.58</td>
<td>69.37</td>
</tr>
<tr>
<td>Seq-transformer, raw programs</td>
<td>99.90</td>
<td>95.32</td>
</tr>
<tr>
<td>Tree2tree LSTM [Chen et al., 2018]</td>
<td>99.76</td>
<td>97.50</td>
</tr>
<tr>
<td>Seq2seq LSTM [Chen et al., 2018]</td>
<td>98.38</td>
<td>12.19</td>
</tr>
</tbody>
</table>

and positional encodings, e.g. the encoding for the second position is summed with the embedding of the first output. This is not an issue in the sequential case; the positional encodings are self-similar, so this “misalignment” is a linear transform away from the “correct” alignment. However, no traversal through a tree’s nodes have this self-similarity property, so proper alignment here is important. We use a zero vector for the start token’s positional encoding, and use the appropriate positional encoding for each autoregressed output.

Our decoder also has to dynamically compute the new positional encoding whenever it produces a token. The decoder must keep track of the partial tree structure that it constructs, to correctly traverse to the next position based on history. It builds this partial tree structure based on which emitted tokens are terminals or non-terminals. In principle, any tree traversal method could be used here, as long as it is used consistently. In our experiments, we explored both depth-first and breadth-first traversals for decoding.

5 Experiments and Results

For our evaluation, we focused on two sets of program translation tasks from the literature to test our model against. The first set of tasks is For2Lam, a synthetic translation dataset between an invented imperative and functional language. The dataset is split into two tasks: one for small programs and one for large programs. The second set of tasks involves translating between generated CoffeeScript and JavaScript code. The data is similarly broken, here both by program length and vocabulary. More details about the data sets can be found at Chen et al. (2018). We report all results in terms of full program accuracy.

As our model expects regular trees, we preprocess all tree data by converting trees to left-child-right-sibling representations, which are binary trees. This enforces \( n = 2 \) for our model. We use a maximum tree depth \( k = 32 \) for all experiments.

5.1 Synthetic Translation Tasks

For the first set of tasks, we trained both our tree-transformers and classic sequence-transformers for baseline experimentation. We trained four models for each task: a sequence-transformer that operates on raw programs; a sequence-transformer that operates on parse tree representations; a tree-transformer with breadth-first traversal; and a tree-transformer with depth-first traversal. Both models were trained with four layers and \( d_{\text{model}} = 256 \). The sequence-transformer was trained with \( d_{ff} = 1024 \) and a positional encoding dimension that matched \( d_{\text{model}} \), in line with the hyperparameters used in the original transformer (Vaswani et al., 2017). The tree-transformer, however, was given a larger positional encoding size of \( 2048 \) in exchange for a smaller \( d_{ff} \) of \( 512 \). This was to emphasize the role of our tree positional encodings, which are inherently bulkier than the sequential positional encodings, while maintaining a similar parameter count. We trained our models with Adam [Kingma & Ba, 2015].

The results for the synthetic tasks can be found in Table 1. Every model besides the seq2seq gets very close to solving the small program dataset, with the depth-first search tree-transformer having a slight advantage. The results on the large program are of more interest: both tree-transformer models perform significantly better than the sequence-transformers, suggesting that the positional
Table 2: Program accuracy data for CoffeeScript-JavaScript translation tasks. While the tree-transformer is slightly disadvantaged on small datasets, possibly due to unstable training, it appears to scale significantly better than LSTM-based models.

encodings help considerably for larger trees. While the breadth-first search variant has very similar performance to the tree2tree LSTM, our depth-first search version achieves the top score by a small margin.

5.2 **COFFEESCRIPT-JAVASCRIPT TRANSLATION**

Given the results on the synthetic tasks, we focused on training depth-first traversal tree-transformers for this task. The data is partitioned four ways, into two sets of vocabulary and two categories of program length. We use the same hyperparameters as in the synthetic tasks, and once again compare our results with the tree2tree and seq2seq models.

The results for CoffeeScript-Javascript translation can be found in Table 2. This data further reinforces that the advantages of the tree-transformer’s design are most prominent with large data. Indeed, while the tree-transformer gains 20 percentage point improvements over the state of the art on the most difficult tasks here, it in fact performs worse than the tree2tree LSTM on two of the smaller tasks. Anecdotally, we noticed some instability during training on small datasets, possibly exacerbated by Adam optimization. Overall, these results are promising for applying tree-transformers to larger-scale scenarios.

6 **CONCLUSION**

We have proposed a novel scheme of custom positional encodings to extend transformers to tree-domain tasks. By leveraging the strengths of the transformer, we have achieved an efficiently parallelizable model that can consider relationships between arbitrary pairs of tree nodes in a single step. Experiments have demonstrated the model’s affinity for tree-structured data, particularly for large tree sizes. We intend to experiment with employing the model on other tree-domain tasks of interest as future work.

By abstracting the transformer’s positional encodings, we have established the potential for generalized transformers to consider other nonlinear structures, given proper implementations. As future work, we are interested in exploring alternative implementations for other domains, in particular graph-structured data as motivated by structured knowledge tasks.

Finally, in this paper we have only considered binary trees: in particular, binary tree representations of trees not originally structured as such. Arbitrary tree representations have their own advantages and complications; we would like to explore training on them directly.

**REFERENCES**


