Insertion-based Tree Decoding

Denis Lukovnikov
Ruhr University Bochum,
Bochum, Germany
denis.lukovnikov@rub.de

Asja Fischer
Ruhr University Bochum,
Bochum, Germany
asja.fischer@rub.de

Abstract

Sequences are typically decoded in a left-to-right fashion, requiring as many decoding steps as there are tokens in the sequence. Recently, several works have proposed non-autoregressive decoders that are sub-linear, allowing to decode a sequence using fewer decoding steps than the length of the sequence, and thus substantially speed up inference. In contrast, non-autoregressive decoding of trees is less well-analysed, even though trees are used in important applications like semantic parsing and code generation. In this work, we present a novel general-purpose partially autoregressive tree decoder that uses tree-based insertion operations to generate trees in sub-linear time. We evaluate our approach on semantic parsing and compare it against strong baselines, including an insertion-based sequence decoder. The results demonstrate that the partially autoregressive tree decoder reaches competitive accuracies while clearly reducing the number of decoding steps.

1 Introduction

Sequence generation is usually based on a left-to-right autoregressive decoder that decomposes the probability of the entire sequence y conditioned on x (x can be empty) as the product $p(y|x) = \prod_{i=0}^{N} p(y_i|y_{<i}, x)$. At each decoding step, the decoder model predicts the next token y_i based on the previously generated outputs $y_{<i}$ and the input x. This approach to decoding sequences is linear in sequence length: the number of decoding steps necessary to produce the sequence is equal to the length of the sequence. However, recently, several works have proposed non-autoregressive decoders that are sub-linear. This allows to decode a sequence using fewer decoding steps than the length of the sequence and can thus greatly speed up inference, especially for longer sequences (Stern et al., 2019; Ma et al., 2019; Ghazvininejad et al., 2019; Gu et al., 2017; Kasai et al., 2020). In particular, the Insertion Transformer of Stern et al. (2019) uses insertion operations to iteratively expand the sequence, achieving a best-case number of $\mathcal{O}(\log_2 N)$ decoding steps.

In this work, we extend insertion-based decoding of sequences to insertion-based decoding of trees. Insertion-based sequence decoder can also be applied to decoding trees (Zhu et al., 2020). However, this requires linearizing trees into sequences, and requires the explicit decoding of subtree termination tokens (e.g. closing parentheses ")"). This results in larger structures, which for the Insertion Transformer increases the minimum necessary number of decoding steps and increases the computational requirements per step. In contrast, the insertion-based tree decoder that we propose here does not need to explicitly decode structure tokens. Moreover, it can achieve a best-case complexity *below* $\mathcal{O}(\log_2 N)$[1] in terms of the number of decoding steps and guarantees that all intermediate outputs are valid trees. To the best of our knowledge, no existing research focuses on insertion-based non-autoregressive decoding of trees so far.

We evaluate the proposed decoder for semantic parsing on the OVERNIGHT (Wang et al., 2015) dataset. Semantic parsing is the task of converting natural language expressions into a formal representation of its meaning. An important application of semantic parsing is question answering from structured data sources. In such use cases, the input to the semantic parser is a natural language question and the expected output is a query, which can be written in a query language (e.g. SQL or SPARQL). Since these queries can often be represented as trees (e.g. abstract syntax tree), semantic parsing is a particularly interesting task for the evaluation of the presented decoding approach.

[1]The exact best possible speed-up heavily depends on the data.

To summarize, the contributions of this work are:

- a transformer-based decoding algorithm that uses insertion operations specifically tailored for decoding trees,
- a novel transformer architecture that uses novel tree-based relative positions,
- and an evaluation of the proposed algorithm and model on the well-known OVERNIGHT dataset, and a comparison against a strong non-autoregressive baseline.

2 Insertion Transformer

Below, we give a very brief overview of the Insertion Transformer, which we use here as a baseline. Due to space constraints, we refer interested readers to the work of Stern et al. (2019) for a more elaborate description of their model and training procedure.

Decoding approach. Rather than decoding autoregressively left-to-right (LTR), the insertion transformer decodes sequences by using insertion operations. For example, consider the sequence "*A B C D E F G*". LTR decoding would require at least seven decoding steps, producing some left-aligned subsequence at every step (e.g. "A B C D" at the fourth step). In contrast, decoding using insertions allows to decode the same sequence using just three steps. Starting from the initial empty state " ", we first decode (1) "***D***", then (2) "***B*** *D* ***F***" and finally (3) "***A*** *B* ***C*** *D* ***E*** *F* ***G***", where the bold faced tokens are the ones inserted in each step, respectively.

Model. The model we use in our experiments uses BERT (Devlin et al., 2019) as the encoder and a standard transformer (Vaswani et al., 2017) with learned absolute position vectors as the decoder. In contrast to the vanilla transformer decoder however, the causal attention mask is not used. After encoding a subsequence using the transformer, the output layer concatenates the representations of two neighbouring tokens to build a representation for the insertion slot between the tokens. We normalize the probabilities per slot.

Training. Given training data consisting of pairs of input and output sequences (x, y), at every epoch, the training algorithm samples a subsequence $\hat{y}$ for every output sequence y. First, a length for the subsequence is drawn from a uniform distribution on $[0, |y|]$. Then, a subsequence $\hat{y}$ of the given length is randomly drawn from y. For example, for the sequence "A B C D E F G" and a randomly drawn length of 3, a sampled subsequence of length 3 could be "B D E".

The Insertion Transformer is then trained by optimizing a loss of the following form:

$$-\sum_{l}\sum_{i=i_l}^{j_l} w_{i,l} \log p(y_i, l | x, \hat{y}) \ , \tag{1}$$

where i_l and j_l are the beginning and end positions of the subsequence to be decoded in slot l. Two variants of this loss function are proposed that differ in the strategy of assigning the weights $w_{i,l}$ to the different tokens and which are referred to as *uniform* and *binary*, respectively. In the uniform case, for a given l the weights $w_{i,l}$, $i = i_l, \dots, j_l$ are equal and sum up to one. In the binary case, a larger weight is assigned to tokens closer to the center of slot l:

$$w_{i,l} = \frac{e^{-d_l(i)/\tau}}{\sum_{i'=i_l}^{j_l} e^{-d_l(i')/\tau}} \ , \tag{2}$$

where $d_l(i)$ is the distance between token i and the center of the span to be decoded in slot l, and τ is the *temperature*.

3 Tree-based Insertion Transformer

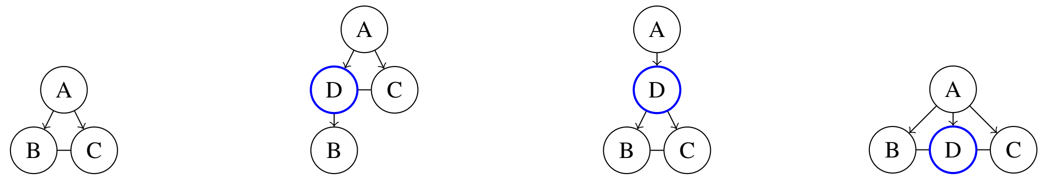
In this section, we propose a novel transformer-based method for non-autoregressive decoding of trees. The proposed method consists of (1) a novel transformer architecture, and (2) a novel insertion-based decoding procedure.

3.1 Decoding

The proposed decoding algorithm is similar to the one proposed by Stern et al. (2019) in that it uses insertion operations to expand the decoded structure. However, instead of using the implicitly defined insertion slots between two neighbouring tokens in a sequence, here, insertion slots are used that are placed between neighbouring nodes in the graph representing the tree.

As an example, consider the tree *(A B C)*[2], shown in Fig. 1a. For sequence-based insertion decoding, we would take the linearization *(· A · B · C ·)*, where four insertion slots are explicitly denoted by "·". The linearized representation clearly has some disadvantages. For example, performing some insertions at some slots can destroy the tree structure,

[2]Trees are given in a Lisp-like notation in this work.



(a) Simple tree (b) Inserting D at the ancestral slot of B. (c) Inserting D at the descendant slot of A. (d) Inserting D at sibling slot between B and C.

Figure 1: Examples of insertions at different insertion slots.

e.g. inserting ")" between B and C yields *(A B) C)*, which is not a valid tree. Also, as mentioned before,

In this work, we propose a tree-based insertion decoding algorithm that defines insertion slots on edges between siblings, as well as on edges between parents and children. Following this approach, the example tree can be described by the following linearization with explicit insertion slots:

$$(\wedge A \vee \mid - \wedge B \vee - \wedge C \vee -) \quad ,$$

where we use three types of insertion slots: (1) ancestor insertion slots denoted by $\wedge$, (2) descendant insertion slots denoted by $\vee$, and (3) sibling insertion slots denoted by –, as described in more detail in the following. Note that the parentheses and the pipe symbol "|" are ignored by the model (when relative positioning is used) and is used mostly for notational and programming convenience. The pipe symbol separates the root portion of the "slotted" subtree string.

Ancestor insertion: If node D is used for the ancestor insertion slot $\wedge$ of *B*, which corresponds to the red slot in

$$(\wedge A \vee \mid - \wedge B \vee - \wedge C \vee -) \quad ,$$

we obtain the tree *(A (D B) C)* (see Fig. 1b). In other words, D replaces B and the entire subtree B is attached as a child of D.

Descendant insertion: If node D is used with a descendant insertion slot $\vee$ of *A*, which corresponds to the red slot in

$$(\wedge A \vee \mid - \wedge B \vee - \wedge C \vee -) \quad ,$$

then the child subtrees of A are moved to node D and the entire subtree D (that now contains the children of A) is attached to A to yield the tree *(A (D B C))*, which is depicted in Fig. 1c.

Sibling insertion: If node D is used with a sibling insertion slot "–" between B and C, which corresponds to the red slot in

$$(\wedge A \vee \mid - \wedge B \vee - \wedge C \vee -) \quad ,$$

then D is inserted as a child of A between B and C to yield the tree *(A B D C)* (see Fig. 1d).

These insertion actions can be used to decode any tree starting from a tree containing only a root node *(ROOT)*. Note that every decoding step is guaranteed to yield a valid tree, unlike in the sequence-based insertion decoder.

3.1.1 Decoder operation

In every step, the decoder takes the previous intermediate tree y_{t-1} (where y_0 = *(ROOT)*) and applies one or more insertion operations to expand the tree to the next intermediate tree y_t. To predict which insertion operations to execute at every available slot, the model described in the following sections encodes the *entire* input tree y_{t-1}. The prediction is then based on this encoding. This procedure is similar to the approach used by Stern et al. (2019). While more efficient methods that re-use computations are possible, we leave an investigation of those for future work. The decoding process is terminated when all slots predict a slot closing actions indicating that no more nodes should be inserted, and thus $y_t = y_{t-1}$.

3.2 Model

We propose a novel architecture that takes advantage of the fact that the intermediate structures generated in all steps of the decoding algorithm are trees. While the encoder stays the same (i.e. BERT), the decoder of our model relies on a transformer-based tagger with tree-based relative positions and a special attention mask. The tree-based relative positions allow us to ignore structure tokens.

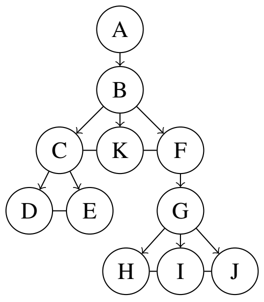


Figure 2: An example of a tree.

3.2.1 Computing Relative Positions in Trees

The proposed architecture uses **tree-based relative positioning** instead of absolute positional embeddings. The relative position is described by the **movements in the tree** that the path from node u to node v defines. For example, in the tree *(A (B (C D E) K (F (G H I J))))* depicted in Fig. 2, the relative position of node D to node I is given by "1↑ 2→ 2↓". This describes the following movement in the tree: starting in D go up one hop to reach parent C (1↑), then move right among the siblings two hops to reach F (2→), and finally move down two hops to reach I (2↓). The scheme is *insensitive to the order of children*, e.g., the relative position from B to K is "1↓", the same as for (B, C) and (B, F). To distinguish these cases, we propose to instead use **special relative position relations** "child_X" and "child_X_of" between parent and child nodes (where X denotes the position of the child among all children of its parent, e.g., X is 1 for node K). While this should improve local modeling of parent-child relations, it still does not add sufficient information to other paths, e.g., the relative positions for (D, H) and (D, J) are identical to that for (D, I)[3].

3.2.2 Using Relative Positions in the Model

With e_{ij} denoting the unnormalized attention scores, x_i the vector corresponding to element i in the input sequence and a_{ij}^K and a_{ij}^V the key (K) and value (V) vector representations of the relative position between the input elements at position i and position j, the following equations describe how the relative position vectors are used in the attention mechanism:

$$e_{ij} = \left(x_i W^Q (x_j W^K)^T + x_i W^Q (a_{ij}^K)^T\right) / \sqrt{d_z}$$
$$z_i = \textstyle\sum_{j=1}^{n} \alpha_{ij} (x_j W^V) + \alpha_{ij} a_{ij}^V \ .$$

This approach to incorporate relative positions is similar to that of Shaw et al. (2018) but differs in the computation of a_{ij}^K's and a_{ij}^V's. A naive implementation would create an independent position embedding for every possible combination of the elements of a movement pattern. For an efficient implementation that is both faster and has fewer parameters, we separately consider the three parts of a movement pattern, embed them separately and add their embeddings:

$$a_{ij}^K = a_{ij}^{K,\uparrow} E^{K,\uparrow} + a_{ij}^{K,\leftrightarrow} E^{K,\leftrightarrow} + a_{ij}^{K,\downarrow} E^{K,\downarrow} \ , \quad (3)$$

where $a_{ij}^{K,\downarrow}$, $a_{ij}^{K,\uparrow}$, and $a_{ij}^{K,\leftrightarrow}$ are one-hot vectors representing the components of the movement pattern, and $E^{K,\downarrow}$, $E^{K,\uparrow}$, and $E^{K,\leftrightarrow}$ are their corresponding embedding lookup matrices.

In case the relative position is not a movement pattern, as is the case for parent-child relations, we simply look up a single embedding vector. We also use relative positions *from* insertion slot positions to some of the node positions: (1) ancestor (∧) and descendant (∨) slots use the "ancestors" and "descendant" relations, respectively, and (2) the sibling slots (–) use the "left_sibling" and "right_sibling" as well as a "parent" relation. Since insertion slot positions are not attended *to* (they are not used as keys in attention), we do not need relative positions *to* slot positions.

3.2.3 Attention Mask

We use a custom attention mask pattern that prevents real query tokens from attending to structural tokens and insertion slots. Since we do insertion-based decoding, we don't use a causal attention mask. However, since structural information is already described by the relative positions (see above), we do not need to process the structure-describing tokens, such as parentheses. Thus, these tokens are masked both as keys and as queries and do not participate in the decoding process at all. Additionally, we use a special mask for the insertion slots (defined above) such that (1) other tokens can not attend *to* the slot tokens and (2) the slot tokens can *only* attend to their immediate neighbours.

3.3 Decoder Training

The decoder learns to imitate optimal trajectories (Section 3.3.1). Let a tree y be the target output

[3]We leave the investigation of better relative position encoding for trees for future work since it is non-trivial to retain constant time complexity of the transformer and have a fully expressive position encoding.

for the current input x. For training the decoder, a **sampling function** (described in Section 3.3.2) is applied to select a partial tree[4] y' from y. A **supervision function** (see Section 3.3.3) is then used that determines the output distribution for every slot, which serves as target in the training of the insertion model. This procedure is detailed in the following.

3.3.1 Optimal Trajectories

The insertion operations described earlier in Section 3.1 can be used to describe a set of actions that transform one partial tree into another. Given an output tree y and the initial tree y_0 that contains a single root node, i.e., y_0 = *(ROOT)*, a *trajectory* from y_0 to y can be defined as a sequence of in total T partial trees (states) $y_0, \ldots, y_T$ and corresponding actions $a_0, \ldots a_T$, such that, when the actions are applied in succession on y_0, they produce $y = y_T$. Each step thus corresponds to the application of an action a_t to the current partial tree y_t resulting in a new partial tree y_{t+1}, i.e. $y_{t+1} = \mathrm{step}(a_t, y_t)$. Each action a_t is a set of atomic actions that insert a node at one of the insertion slots, which can be either ancestor, descendant, or sibling insertion slots. The atomic actions are described as tuples (k, w), where k is the slot in y_t where a node with label w will be inserted when the action is executed. Note that the type of insertion is characterized by the type of insertion slot.

While many trajectories exists that successfully reach y from y_0, we are interested in *optimal trajectories*, which are trajectories that minimize the number of decoding steps T that need to be performed.

Computing Optimal Trajectories: To practically compute trajectories where we try to minimize the number of steps taken, we rely on the fact that first decoding the most central nodes of a slot's subgraphs enables greater parallelization. For a given tree y_t at decoding step t that is a partial tree of the original tree y, we (1) align y_t and y and determine which nodes are allowed to be inserted in every slot in y_t and (2) compute which of these nodes is the best in order to minimize the number of decoding steps.

[4]Note that what we refer to as a partial tree is not the same as a subtree. A subtree retains all the descendants starting from a certain parent node. In contrast, we refer with partial tree to any tree consisting of nodes that can also be found in the original tree, and which can be extended to the full tree y by means of the defined insertion operations.

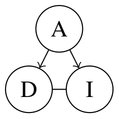


Figure 3: A partial tree of the tree in Fig. 2.

Computing allowed insertions: Given a partial tree y_t (e.g. Fig. 3) aligned with the original tree y (e.g. Fig. 2), we first compute the set of allowed insertions C_k for every slot k.

For an **ancestor insertion slot** ($\wedge$) in y_t associated with some node n, C_k corresponds to the nodes from y on the path from n up to the *lowest used ancestor* of n. The lowest used ancestor $\mathrm{lua}(n)$ of a node n from the partial tree y_t is the lowest[5] node in the original tree y that is an ancestor of n as well as *any* other node from y_t. That is, for node I in the example tree in Fig. 2, the lowest used ancestor of node I is the node B: $\mathrm{lua}(I) = B$ and the set of allowed nodes for I's ancestor slot is $\{F, G\}$.

For a **descendant insertion slot** ($\vee$), the set of allowed nodes C_k is the set of all descendants of n in y , if n doesn't have children in the partial tree y_t. Otherwise, it's the set of all of its children in y_t that are also an ancestor for *all* the children of n in the partial tree y_t.

Finally, for a **sibling insertion slot** (–), to find the set of allowed nodes, we first find the lowest common ancestor in the original tree y of the slot's left node l and right node r. The lowest common ancestor $\mathrm{lca}(n, n')$ of two nodes n and n' in a tree is the lowest node that is an ancestor of both n and n'. We first determine the set C of children of $\mathrm{lca}(l, r)$ that are between the children of $\mathrm{lca}(l, r)$ and that are also the ancestors of nodes l or r. The set of allowed nodes C_k for this slot is then the set C as well as all their descendants in y. In the example, the sibling insertion slot between D and I should accept only the node K since $\mathrm{lca}(D, I)$ is B and K is the only node between C and F, which are the ancestors of D and I, respectively, that are also children of $\mathrm{lca}(D, I)$. [6]

Computing best insertions: Now, for each insertion slot k in y_t, we are given the set C_k of nodes allowed to be inserted. The node $n \in C_k$

[5]Lowest and highest refer to tree depth, i.e., the root node is the highest node in the tree.

[6]Note that inserting E between D and I will lead to a tree from which we can't recover the original tree since D, E and I now assume the same parent and there is no action defined to separate them under different parents.

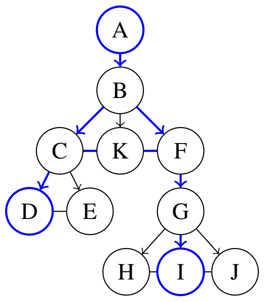


Figure 4: The tree from Fig. 2 and the nodes and edges covered by the partial tree from Fig. 3. Nodes with the same labels in Figures 2 and 3 are aligned.

to be inserted in slot k in order to minimize the number of decoding steps is the most central node in C_k.

The centrality of a node is computed over the subgraph G_k of the original tree y that contains the nodes in C_k, as well as their descendants that are not separated by the partial tree. The *closeness centrality* is used:

$$H_{G_k}(n) = \frac{1}{\sum_{n' \in G_k} d(n, n')} \quad , \tag{4}$$

where $d(n, n')$ is the distance between nodes n and n' in the original tree, which is the minimum number of steps necessary to reach n from n'.

The node picked to be inserted into some slot k is then the one with highest centrality for the slot.

Computing best actions: The best action for a certain partial tree y_t of y then consists of the insertions that are the best for every of its insertion slots. If C_k is empty for some insertion slot, a dummy insertion operation is used that does not insert anything.

3.3.2 Partial Tree Sampling

Rather than sampling all possible partial trees, which would be equivalent to the method described by Stern et al. (2019), we use a different method that samples only from the most optimal trajectory. The partial trees that are used for training are only those that occur on one of the optimal trajectories. For efficiency reasons, we precompute a certain number (5 in our experiments) of trajectories, where we randomly sample when ties occur in the centrality measure, and reuse these trajectories throughout training.[7] Sampling more than one trajectory could reduce the exposure bias.

[7]Note that only one of the partial trees y_t of some example is used in a single epoch.

3.3.3 Supervising Partial Trees

To produce the target distribution for a slot k, we take the nodes $n \in C_k$ computed as valid insertions, as well as their centralities. Then, we rank the nodes in C_k by centrality scores, where the most central node is the highest-ranked one, receiving rank value 0. Ties in centrality are broken by favouring nodes that are lower in the tree y and further ties are broken alphabetically (based on node label). The target distribution for a slot k is then computed using a softmax:

$$p_k(n) = \frac{e^{-\mathrm{rank}_k(n)/\tau}}{\sum_{n' \in C_k} e^{-\mathrm{rank}_k(n')/\tau}} \quad , \tag{5}$$

where τ is a temperature hyperparameter. $p_k(n)$ is zero if n is not in C_k and $\mathrm{rank}_k(n)$ is rank value given to node n.

For each slot k the model outputs a predictive distribution $\pi_k(n)$. Given the target distributions for all slots the training loss is the sum of the Kullback-Leibler (KL) divergences between the target and predictive distributions for all slots:

$$-\sum_k \sum_{n \in \mathcal{G}} p_k(n) \log \frac{\pi_k(n)}{p_k(n)} \quad , \tag{6}$$

where $\mathcal{G}$ denotes the set of all possible node labels.

4 Experiments[8]

We run experiments on the OVERNIGHT (Wang et al., 2015) dataset and report the results in Tables 1 and 2. A description of the dataset and its statistics are provided in Appendix A.

Evaluation: The metric reported is logical form accuracy, which we compute by (1) taking the predictions of the models, (2) balancing parentheses on the left and on the right and (3) computing whether the trees are the same.[9]

Data preprocessing: We notice that many examples in the Overnight dataset contain nested filters. This fact was ignored during training and evaluation of some models in previous work (Damonte

[8]The code is provided at https://github.com/lukovnikov/parseq/tree/crforen/parseq/scripts_insert

[9]The trees are considered the same if every node's children occur in both trees in the same order, if the node's children should be ordered (for example argmax expressions) and if every node's children occur in both trees in *any* order for nodes of which the children should not be ordered (for example SW:concat and a collection of filter conditions).

et al., 2019; Xu et al., 2020). During evaluation, an example was considered wrong if the filters were decoded in a different nesting order. For this reason, we use slightly simplified logical forms for the Overnight dataset that remove nesting between filters (see also Appendix B). Evaluation with these logical forms better reflects the true meaning of the queries. See Appendix C for an example.

For the sequence-based decoder, we found that it is necessary to use **numbered tokens** to to successfully train the model. We simply replace every token "X" by a token "X_d", where d specifies how many tokens "X" have been observed before and including the current token "X". For example, this would transform "(A (B C) (B D))" into "$(_1$ A_1 $(_2$ B_1 C_1 $)_1$ $(_3$ B_2 D_1 $)_2$ $)_3$". This ensures that every token in the sequence is unique. We use numbered tokens in all the experiments, both for the sequence-based as well as the tree-based decoders. We leave a deeper investigation of the effect of numbering tokens for future work.

Baselines: We considered the following models as baselines in our experiments: (1) BERT with a transformer decoder (BERT+Transformer), (2) a re-implemented insertion transformer (Seq-Insert) in both binary and uniform supervision settings (BERT is used for encoding the input), and (3) BERT with a regular tree-based decoder similar to the one proposed by Dong and Lapata (2016) (BERT+TreeGRU; see also Appendix E). Note that both (1) and (3) are essentially left-to-right decoders and can't decode in parallel. Even though they are not directly comparable, we also include the results reported by Chen et al. (2018a). Rather than decoding logical forms (i.e. Lisp-style expressions), they decode an intermediate representation called a query graph using special graph generation actions (e.g. add variable node, add edge). Moreover, the evaluation is based on execution accuracy instead of logical form accuracy (Xu et al., 2020)

Training details: We train all models using Adam (Kingma and Ba, 2014), varying the initial learning rate within $\{0.0001, 0.00005, 0.00001\}$ and experimenting with dropout rates out of $\{0.0, 0.1, 0.2, 0.4\}$. We also experimented with values from $\{1.0, 0.1\}$ for the temperature τ. The validation set for each domain was constructed by taking a random 20% subset of the training examples. This validation set was used for early stopping. We randomly searched the hyperparameter space and took the best performing parameters based on its validation performance on the *publications* domain. These hyperparameters[10] were used for training the models on the other domains. We train each model three times on every domain independently (with the same seed values shared over domains and settings) and also report the average over the domains. For all our experiments, we use a 12-layer BERT model from Huggingface (Wolf et al., 2020) as the encoder and use a transformer with 6 layers, 12 heads, and 768 dimension for the decoder.

4.1 Results

Table 1 shows the results on all domains of the Overnight dataset. Table 2 shows the results on four Overnight domains, and indicates the accuracy and speed-up measured in terms of the number of decoding steps compared to the BERT+Transformer left-to-right baseline.

Baseline equivalence: First, we establish that the results of our left-to-right baseline are on par with previous reported numbers in similar settings. This is shown in the middle part of Table 1, where our "BERT+Transformer" beats Xu et al. (2020)'s BERT+LSTM baseline by a small margin.

Sequence-based insertion baseline results: In the bottom part of Table 1, we report numbers for the slightly simplified logical forms. The performance of the parallel sequence insertion decoder with binary supervision is on par with that of our left-to-right baseline. The uniform variant performs slightly worse. Concerning the speed-up obtained, Table 2 shows that the binary supervision is (more than $\times 2$) more effective than uniform supervision. However, even the binary supervised version lags behind the theoretically best possible speed-up ("Seq-Insert Th.B.").

Tree-based insertion results: The tree-based insertion decoding procedure described in Section 3.1 enables a decoding complexity that is lower than the theoretically best $\mathcal{O}(log_2(N))$ of the sequence-based insertion decoder. However, the theoretically best possible speed-up for tree insertion decoding heavily depends on the tree structure. We report this number ("Tree-Insert Th.B.") for the different domains in Table 2. The theoretically best possible speedup of the tree-based insertion decoder is

[10] dropout rate=0.2, learning rate=0.00005, τ=0.1, batch size=10/30/50, cosine learning rate scheduler with 20 epoch warm-up, 200 epochs

	cal.	blo.	hou.	res.	pub.	rec.	soc.	bas.	avg (± std)
Seq2Action (Chen et al., 2018b)	81.5	61.4	74.1	80.7	80.7	82.9	82.1	88.2	79.0
Shift-Reduce (Damonte et al., 2019)	43.5	25.1	29.6	37.3	32.9	58.3	51.2	69.6	43.4
BERT+LSTM (Xu et al., 2020)	58.3	42.6	48.7	55.4	64.6	68.5	70.4	84.1	61.6
BERT+Transformer	61.3	45.4	50.8	58.7	63.4	76.4	69.9	85.4	63.9
BERT+Transformer	80.0	53.9	70.9	83.4	70.4	83.3	73.8	84.0	75.0 ± 1.3
BERT+TreeGRU	77.4	49.7	67.9	82.6	73.3	81.2	73.0	84.4	73.7 ± 1.5
Seq-Insert (Binary)	78.2	50.9	67.5	81.0	72.5	81.2	70.9	84.1	73.3 ± 1.6
Seq-Insert (Uniform)	79.0	47.8	68.3	80.5	70.0	82.3	70.0	83.7	72.7 ± 1.6
Tree-Insert	77.4	51.9	71.8	81.1	72.9	82.9	73.2	84.4	74.4 ± 1.3

Table 1: Results on the test set of the different domains of the Overnight dataset. Top part: denotation accuracy. Middle part: logical form accuracy on original trees. Bottom part: logical form accuracy on simplified trees. In the last column, we report the average accuracy over the domains as well as the average (over domains) of standard deviations of accuracies for every domain over different seeds.

slightly higher than that of the sequence-based insertion decoder: an average of 0.48 decoding steps can be gained, which corresponds to a potential reduction of decoding steps of $\times 1.08$ on average. However, one needs to investigate how close an actual tree-based insertion decoder can get to the theoretically best number in practice.

The bottom part of Table 2 reports the results for the tree-based insertion decoder. Our insertion-based tree decoder achieves **competitive accuracy** to both the left-to-right baseline as well as the sequence insertion decoder, while requiring **fewer decoding steps** than the strong sequence insertion baseline: an average of 0.7 decoding steps is gained, which corresponds to an average of a $\times 1.11$ reduction in the number of decoding steps.

Ablation study: In Table 3, we assess the effects of some design choices and hyperparameters. Decreasing the number of trajectories used during training from 5 to 1 results in a significant decrease in accuracy. The chosen temperature τ also affects training. When τ is set to a high value, the target distribution becomes more uniform, which allows the model to use insertions that are not covered by the trained trajectories making it more likely to fail due to exposure bias, which is reflected in the poor result for $\tau = 10$. Using absolute instead of relative position information leads only to a slight decrease in accuracy. Note that using just absolute positioning requires to process the structure tokens because the structure information contained in relative positions is not being used. The child relations (§3.2.2) appear to not have any significant effect.

5 Discussion

To obtain an efficient decoder other considerations next to the number of decoding steps must be taken into account. The actual execution speed and computational load also heavily depend on (1) the size of processed data and (2) efficiency of implementation. In the proposed tree decoder, the *effective* size of model inputs is *smaller* than for the sequence insertion decoder since it does not need parentheses. Thus, when implemented efficiently, it requires less computation and less memory, which is attractive given the quadratic memory complexity of the transformer's attention.

Another point worth noting is that the best possible speed-up of the proposed insertion-based tree decoder heavily depends on the data. However, considering that the proposed model has between two and three insertion slots per token, and the sequence insertion decoder only one, our method can expand trees much faster.

A limitation of the proposed insertion-based tree decoder is that it, like the Insertion Transformer, is unable to recover from mistakes made during decoding. In contrast, the Levenshtein transformer (Gu et al., 2019b) also defines deletion operations and can thus recover from erroneous predictions. Extending the insertion-based tree decoder to also delete part of the trees is an interesting direction for future work. While we simply sample from a number of optimal trajectories, a more general decoder that allows for deletion would have to be trained using dynamic oracles (Goldberg and Nivre, 2012; Ross et al., 2011; Vlachos and Clark, 2014) or reinforcement learning. This is likely to

	calendar		restaurants		publications		recipes		average	
	Acc.	# Steps	Acc.	# Steps	Acc.	# Steps	Acc.	# Steps	Acc.	# St.
BERT+Transf.	80.0	40.4 (×1.0)	83.4	37.8 (×1.0)	70.4	40.9 (×1.0)	83.3	36.6 (×1.0)	79.3	38.9
BERT+TreeGRU	77.4	41.2 (×1.0)	82.6	38.1 (×1.0)	73.3	41.3 (×1.0)	81.2	36.7 (×1.0)	78.6	39.3
Seq-Insert Th.B.	–	6.7 (×6.0)	–	6.6 (×5.7)	–	6.7 (×6.1)	–	6.5 (×5.6)	–	6.6
Seq-Insert (Bin.)	78.2	7.2 (×5.6)	81.0	6.9 (×5.5)	72.5	7.3 (×5.6)	81.2	6.9 (×5.3)	78.2	7.1
Seq-Insert (Uni.)	79.0	18.3 (×2.2)	80.5	16.9 (×2.2)	70.0	17.6 (×2.3)	82.3	16.8 (×2.2)	77.9	17.4
Tree-Insert Th.B.	–	6.2 (×6.5)	–	6.1 (×6.2)	–	6.3 (×6.5)	–	6.0 (×6.1)	–	6.2
Tree-Insert	77.4	6.4 (×6.4)	81.1	6.2 (×6.2)	72.9	6.6 (×6.3)	82.9	6.2 (×5.8)	78.6	6.4

Table 2: Logical form accuracy on simplified logical forms and number of decoding steps used on the test set of the different domains of the Overnight dataset. “Bin” stands for binary supervision, “Uni.” for uniform.

Ablation	publications	calendar	recipes
$\tau = 1.$	69.8 (-3.1)	74.0 (-3.4)	82.3 (-0.6)
$\tau = 10.$	24.4 (-48.5)	20.0 (-57.4)	28.9 (-54.0)
# traj. = 1	67.3 (-5.6)	71.6 (-5.8)	66.5 (-16.4)
abs. pos	71.0 (-1.9)	77.0 (-0.4)	79.3 (-3.6)
no child. rel.	72.5 (-0.4)	78.8 (+1.4)	81.9 (-1.0)

Table 3: Results of an ablation study on the PUBLICATIONS domain.

decrease the exposure bias as the model is exposed to many different trajectories.

6 Related Work

To the best of our knowledge, to this date, only two other works have investigated non-autoregressive methods for trees and semantic parsing. The work of Rubin and Berant (2020) proposes a bottom-up tree decoder, however, their decoder is limited to being linear in depth and thus less parallelizable for deeper and narrower trees. In contrast, the Insertion Transformer can be sub-linear both in depth and breadth. Zhu et al. (2020) apply the Insertion Transformer to semantic parsing, albeit on SNIPS (Coucke et al., 2018), ATIS (Price, 1990) and TOP (Gupta et al., 2018) datasets and focuses on cross-lingual performance.

Only a few transformer architectures specialized for trees have been proposed to date. The work of Shiv and Quirk (2019) proposes a new positional encoding to improve tree representation using transformers. The work of Anonymous (2020), similarly to ours, investigates relative positioning for transformers operating on trees. Other architectures to encode trees exist as well, such as the TreeLSTM (Tai et al., 2015), however, a transformer-based model enjoys greater parallelism and direct modeling of long-range dependencies.

Several methods have recently been proposed for non-autoregressive decoding and insertion-based decoding. Stern et al. (2019), Gu et al. (2019a) and Gu et al. (2019b) experiment with insertion-based decoding where Gu et al. (2019b) also support deletion operations. Ma et al. (2019), on the other hand, develop a non-autoregressive sequence generation model using normalizing flows (Rezende and Mohamed, 2015). Some other examples of non-autoregressive decoding for NMT are (Ghazvininejad et al., 2019; Gu et al., 2017; Kasai et al., 2020).

Various works have recently explored the use of neural networks for semantic parsing. Dong and Lapata (2016) explore both sequence-based as well as tree-structured decoding. Alvarez-Melis and Jaakkola (2017) propose an improved tree decoder that uses additional classifiers to perform structure prediction rather than using structure tokens. Chen et al. (2018c) propose a binary tree generator for code translation.

7 Conclusion

In this work, we presented a novel neural network based method for non-autoregressive decoding of trees. We define insertion operations used in the step-wise decoding process that guarantee that intermediate structures are trees. This results in a reduction of the number of decoding steps and allows to exploit tree structure. Experiments on semantic parsing show competitive accuracy and a significantly decreased number of decoding steps, compared to strong autoregressive and non-autoregressive baselines.

Acknowledgments

We thank the reviewers for their helpful comments and suggestions. We also thank Prof. Dr. Jens Lehmann, Priyansh Trivedi, and Gaurav Maheshwari for their comments and help with this work.

References

David Alvarez-Melis and Tommi S Jaakkola. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*.

Anonymous. 2020. A structural transformer with relative positions in trees for code-to-sequence tasks.

Bo Chen, Le Sun, and Xianpei Han. 2018a. Sequence-to-action: End-to-end semantic graph generation for semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777, Melbourne, Australia. Association for Computational Linguistics.

Bo Chen, Le Sun, and Xianpei Han. 2018b. Sequence-to-action: End-to-end semantic graph generation for semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777.

Xinyun Chen, Chang Liu, and Dawn Song. 2018c. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 2552–2562.

Alice Coucke, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, et al. 2018. Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces.

Marco Damonte, Rahul Goel Tagyoung Chung, and Amazon Alexa AI. 2019. Practical semantic parsing for spoken language understanding. *NAACL HLT 2019*, pages 16–23.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. Mask-predict: Parallel decoding of conditional masked language models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6114–6123.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. 2017. Non-autoregressive neural machine translation. *arXiv preprint arXiv:1711.02281*.

Jiatao Gu, Qi Liu, and Kyunghyun Cho. 2019a. Insertion-based decoding with automatically inferred generation order. *Transactions of the Association for Computational Linguistics*, 7:661–676.

Jiatao Gu, Changhan Wang, and Junbo Zhao. 2019b. Levenshtein transformer. In *Advances in Neural Information Processing Systems*, volume 32, pages 11181–11191. Curran Associates, Inc.

Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. 2018. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2787–2792.

Jungo Kasai, James Cross, Marjan Ghazvininejad, and Jiatao Gu. 2020. Parallel machine translation with disentangled context transformer. *arXiv preprint arXiv:2001.05136*.

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

Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard Hovy. 2019. Flowseq: Non-autoregressive conditional sequence generation with generative flow. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4273–4283.

Patti Price. 1990. Evaluation of spoken language systems: The atis domain. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990.*

Danilo Rezende and Shakir Mohamed. 2015. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538. PMLR.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings.

Ohad Rubin and Jonathan Berant. 2020. Smbop: Semi-autoregressive bottom-up semantic parsing. *arXiv preprint arXiv:2010.12412*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468.

Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *NeurIPS*, pages 12058–12068.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. In *International Conference on Machine Learning*, pages 5976–5985.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566.

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

Andreas Vlachos and Stephen Clark. 2014. A new corpus and imitation learning framework for context-dependent semantic parsing. *Transactions of the Association for Computational Linguistics*, 2:547–560.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Silei Xu, Sina Semnani, Giovanni Campagna, and Monica Lam. 2020. Autoqa: From databases to q&a semantic parsers with only synthetic training data. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 422–434.

Qile Zhu, Haidar Khan, Saleh Soltan, Stephen Rawls, and Wael Hamza. 2020. Don't parse, insert: Multilingual semantic parsing with insertion based decoding. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 496–506.

A Dataset Description and Statistics

The OVERNIGHT dataset consists of pairs of natural language questions and corresponding formal language queries from 8 distinct domains, such as "publications" and "restaurants". The dataset was generated as follows: (1) first, a grammar was defined, which consists of a general part applicable for any domain, and a domain-specific part that specifies a *seed lexicon* mapping between predicates and NL. Then, (2) a number of examples was generated, which at this point consist of (i) a canonical utterance and (ii) a formal query. Finally, (3) the canonical utterances are paraphrased by Amazon Mechanical Turk workers to generate more natural examples. See Appendix C for an example question-query pair from the data set.

See Table 4 for dataset statistics.

Domain	#Train	#Test	Avg. tree size	Avg. tree depth	Avg. seq. length
cal.	669	168	17.5	5.7	39.1
blo.	1596	399	18.8	5.8	42.1
hou.	752	189	17.3	5.6	38.8
res.	1325	332	16.3	5.4	37.1
pub.	640	161	18.3	5.8	41.1
rec.	864	216	16.1	5.7	36.3
soc.	3535	884	25.3	8.7	57.3
bas.	1561	391	19.8	8.2	45.1

Table 4: Statistics for the OVERNIGHT dataset after simplification.

B Preprocessing

This simplified version is obtained by (1) merging the "call" node with the function of the call (e.g. replacing *(call SW:someFunction ...)* with *(call-SW:someFunction ...))* and (2) merging nested filters into a multi-conditional expression where the order of conditions doesn't matter. Note that these simplifications are easily reversible. During training, we sort the different parts of a multi-conditional filter expression alphabetically and during evaluation, we consider the different conditions as unordered and accept them in any decoded order. We use this simplified version for all further experiments.

C An example from the Overnight dataset

An actual example from the "publications" domain from the OVERNIGHT dataset is the following.

The natural language question is:

"*find an article published in 2004*".

The formal query corresponding to this question is (after preprocessing):

Listing 1: Example query

```
(call_SW_listValue
   (filter
      (call_SW_getProperty
         (call_SW_singleton
            (en.article ))
         (string
            (!type )))
      (condition
         (string
            (publication_date ))
         (string
            (= ))
         (date
            (2004 )
            (−1 )
            (−1 )))))
```

D An example of insertion-based decoding for a real example.

Consider the example in Appendix C. This example can be decoded in 5 steps using tree-based insertion actions as elaborated in the following. Note that multiple insertion action sequences would work with the same number of steps.

Listing 2: Example query

```
1.    (filter)
```

Listing 3: Example query

```
2. (call_SW_listValue
      (filter
            (string)))
```

Listing 4: Example query

```
3. (call_SW_listValue
      (filter
            (call_SW_singleton)
         (condition
            (string)
         (string
            (=  ))
         (date))))
```

Listing 5: Example query

4.
   ```
   (call_SW_listValue
      (filter
         (call_SW_getProperty
            (call_SW_singleton
               (en.article  ))
            (string))
         (condition
            (string
               (publication_date  ))
            (string
               (=  ))
            (date
               (-1  )))))
   ```

5. Same as original tree in Appendix C.

E TreeGRU

We implement the tree-based left-to-right baseline decoder similarly to Dong and Lapata (2016). However, in our implementation, we use depth-first instead of breadth-first decoding. We achieve similar conditioning in the different decoding steps by manipulating which of the previous states are used as the previous sibling state and the parent state. If the previously decoded token was an opening parenthesis "(", then the parent state correspond to the state of the GRU after producing the opening parenthesis (this is the previous state), the previous sibling state is set equal to the parent state. Following Dong and Lapata (2016), we use the parent state explicitly as part of the GRU input in every step. When a closing parenthesis ")" is decoded, the parent of the previous parent state is used as the parent state, the state accumulated from the closed subtree is discarded, and the previous parent state becomes the previous sibling state. Thus, we make sure that similarly to Dong and Lapata (2016)'s breadth-first decoding, the different branches aren't directly conditioned on each other through decoder states.