Exploring Dynamic Selection of Branch Expansion Orders for Code Generation

Hui Jiang[1,2*] **Chulun Zhou**[1,2*] **Fandong Meng**[3] **Biao Zhang**[4] **Jie Zhou**[3]
Degen Huang[5] **Qingqiang Wu**[1,2] **Jinsong Su**[1,2†]
[1]School of Informatics, Xiamen University
[2]Institute of Artificial Intelligence, Xiamen University
[3]Pattern Recognition Center, WeChat AI, Tencent Inc, China
[4]School of Informatics, University of Edinburgh [5]Dalian University of Technology
{hjiang,clzhou}@stu.xmu.edu.cn {fandongmeng,withtomzhou}@tencent.com
B.Zhang@ed.ac.uk huangdg@dlut.edu.cn {wuqq,jssu}@xmu.edu.cn

Abstract

Due to the great potential in facilitating software development, code generation has attracted increasing attention recently. Generally, dominant models are Seq2Tree models, which convert the input natural language description into a sequence of tree-construction actions corresponding to the pre-order traversal of an Abstract Syntax Tree (AST). However, such a traversal order may not be suitable for handling all multi-branch nodes. In this paper, we propose to equip the Seq2Tree model with a context-based *Branch Selector*, which is able to dynamically determine optimal expansion orders of branches for multi-branch nodes. Particularly, since the selection of expansion orders is a non-differentiable multi-step operation, we optimize the selector through reinforcement learning, and formulate the reward function as the difference of model losses obtained through different expansion orders. Experimental results and in-depth analysis on several commonly-used datasets demonstrate the effectiveness and generality of our approach. We have released our code at https://github.com/DeepLearnXMU/CG-RL.

1 Introduction

Code generation aims at automatically generating a source code snippet given a natural language (NL) description, which has attracted increasing attention recently due to its potential value in simplifying programming. Instead of modeling the abstract syntax tree (AST) of code snippets directly, most of methods for code generation convert AST into a sequence of tree-construction actions. This allows for using natural language generation (NLG) models, such as the widely-used encoder-decoder models, and obtains great success (Ling et al., 2016; Dong and Lapata, 2016, 2018; Rabinovich et al., 2017; Yin and Neubig, 2017, 2018, 2019; Hayati et al., 2018; Sun et al., 2019, 2020; Wei et al., 2019; Shin et al., 2019; Xu et al., 2020; Xie et al., 2021). Specifically, an encoder is first used to learn word-level semantic representations of the input NL description. Then, a decoder outputs a sequence of tree-construction actions, with which the corresponding AST is generated through pre-order traversal. Finally, the generated AST is mapped into surface codes via certain deterministic functions.

Generally, during the generation of dominant Seq2Tree models based on pre-order traversal, branches of each multi-branch nodes are expanded in a left-to-right order. Figure 1 gives an example of the NL-to-Code conversion conducted by a Seq2Tree model. At the timestep t_1, the model generates a multi-branch node using the action a_1 with the grammar containing three fields: *type*, *name*, and *body*. Thus, during the subsequent generation process, the model expands the node of t_1 to sequentially generate several branches in a left-to-right order, corresponding to the three fields of a_1. The left-to-right order is a conventional bias for most human-beings to handle multi-branch nodes, which, however, may not be optimal for expanding branches. Alternatively, if we first expand the field *name* to generate a branch, which can inform us the name 'e', it will be easier to expand the field *type* with a 'Exception' branch due to the high co-occurrence of 'e' and 'Exception'.

To verify this conjecture, we choose TRANX (Yin and Neubig, 2018) to construct a variant: TRANX-R2L, which conducts depth-first generation in a right-to-left manner, and then compare their performance on the DJANGO dataset. We find that about 93.4% of ASTs contain multi-branch nodes, and 17.38% of AST nodes are multi-branch

Joint work with Pattern Recognition Center, WeChat AI, Tencent Inc, China.
*Equal contribution
†Corresponding author

Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pages 5076–5085
August 1–6, 2021.

	Percentage
Only TRANX	8.47
Only TRANX-R2L	7.66

Table 1: The percentages of multi-branch nodes, which can only be correctly handled by different models. TRANX-R2L is a variant of TRANX (Yin and Neubig, 2018), which handles multi-branch nodes in a right-to-left order.

ones. Table 1 reports the experimental results. We can observe that 8.47% and 7.66% of multi-branch nodes can only be correctly handled by TRANX and TRANX-R2L, respectively. Therefore, we conclude that different multi-branch nodes have different optimal branch expansion orders, which can be dynamically selected based on context to improve the performance of conventional Seq2Tree models.

In this paper, we explore dynamic selection of branch expansion orders for code generation. Specifically, we propose to equip the conventional Seq2Tree model with a context-based *Branch Selector*, which dynamically quantifies the priorities of expanding different branches for multi-branch nodes during AST generations. However, such a non-differentiable multi-step operation poses a challenge to the model training. To deal with this issue, we apply reinforcement learning to train the extended Seq2Tree model. Particularly, we augment the conventional training objective with a reward function, which is based on the model training loss between different expansion orders of branches. In this way, the model is trained to determine optimal expansion orders of branches for multi-branch nodes, which will contribute to AST generations.

To summarize, the major contributions of our work are three-fold:

- Through in-depth analysis, we point out that different orders of branch expansion are suitable for handling different multi-branch AST nodes, and thus dynamic selection of branch expansion orders has the potential to improve conventional Seq2Tree models.
- We propose to incorporate a context-based *Branch Selector* into the conventional Seq2Tree model and then employ reinforcement learning to train the extended model. To the best of our knowledge, our work is the first attempt to explore dynamic selection of branch expansion orders for code generation.
- Experimental results and in-depth analyses demonstrate the effectiveness and generality of our model on various datasets.

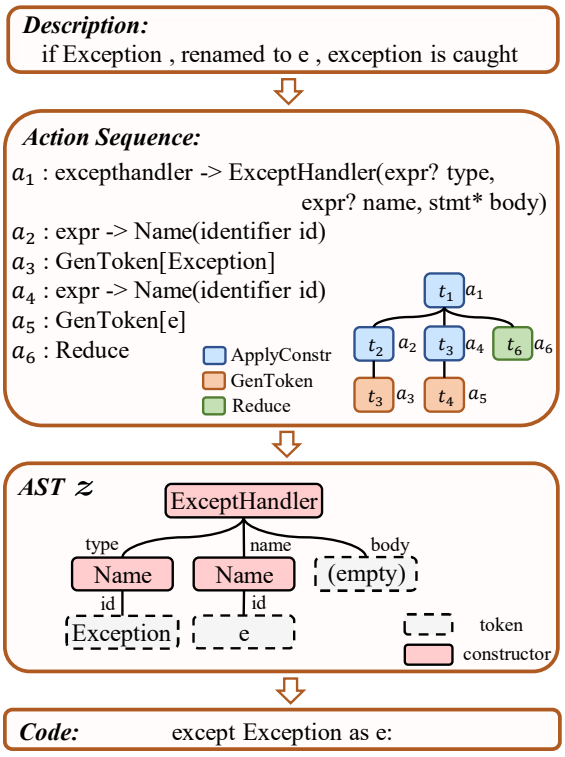


Figure 1: An example of code generation using the conventional Seq2Tree model in pre-order traversal.

2 Background

As shown in Figure 1, the procedure of code generation can be decomposed into three stages. Based on the learned semantic representations of the input NL utterance, the dominant Seq2Tree model (Yin and Neubig, 2018) first outputs a sequence of abstract syntax description language (ASDL) grammar-based actions. These actions can then be used to construct an AST following the pre-order traversal. Finally, the generated AST is mapped into surface code via a user-specified function AST_to_MR($*$).

In the following subsections, we first describe the basic ASDL grammars of Seq2Tree models. Then, we introduce the details of TRANX (Yin and Neubig, 2018), which is selected as our basic model due to its extensive applications and competitive performance (Yin and Neubig, 2019; Shin et al., 2019; Xu et al., 2020). [1]

[1]Please note that our approach is also applicable to other Seq2Tree models.

2.1 ASDL Grammar

Formally, an ASDL grammar contains two components: *type* and *constructors*. The value of *type* can be composite or primitive. As shown in the '$ActionSequence$' and '$ASTz$' parts of Figure 1, a constructor specifies a language component of a particular type using its fields, e.g., *ExceptHandler (expr? type, expr? name, stmt∗ body)*. Each field specifies the type of its child node and contains a cardinality (single, optional ? and sequential ∗) indicating the number of child nodes it holds. For instance, *expr? name* denotes the field *name* has optional child node. The field with composite type (e.g. *expr*) can be instantiated by constructors of corresponding type, while the field with primitive type (e.g. *identifier*) directly stores token.

There are three kinds of ASDL grammar-based actions that can be used to generate the action sequence: 1) **APPLYCONSTR[c]**. Using this action, a constructor c is applied to the composite field of the parent node with the same type as c, expanding the field to generate a branch ending with an AST node. Here we denote the field of the parent node as *frontier field*. 2) **REDUCE**. It indicates the completion of generating branches for a field with optional or multiple cardinalities. 3) **GENTOKEN[v]**. It expands a primitive frontier field to generate a token v.

Obviously, a constructor with multiple fields can produce multiple AST branches[2], of which generation order has important effect on the model performance, as previously mentioned.

2.2 Seq2Tree Model

Similar to other NLG models, TRANX is trained to minimize the following objective function:

$$\mathcal{L}_{mle}(\boldsymbol{x}, \boldsymbol{a}) = -\sum_{t=1}^{T} \log p(a_t | a_{<t}, \boldsymbol{x}), \quad (1)$$

where a_t is the t-th action, and $p(a_t|a_{<t}, \boldsymbol{x})$ is modeled by an attentional encoder-decoder network (Yin and Neubig, 2018).

For an NL description $\boldsymbol{x}$=$x_1, x_2, ..., x_N$, we use a BiLSTM encoder to learn its word-level hidden states. Likewise, the decoder is also an LSTM network. Formally, at the timestep t, the temporary hidden state $\mathbf{h}_t$ is updated as

$$\mathbf{h}_t = f_{\text{LSTM}}\left([\mathbf{E}(a_{t-1}) : \mathbf{s}_{t-1} : \mathbf{p}_t], \mathbf{h}_{t-1}\right), \quad (2)$$

where $\mathbf{E}(a_{t-1})$ is the embedding of the previous action a_{t-1}, $\mathbf{s}_{t-1}$ is the previous decoder hidden state, and $\mathbf{p}_t$ is a concatenated vector involving the embedding of the frontier field and the decoder hidden state for the parent node. Furthermore, the decoder hidden state $\mathbf{s}_t$ is defined as

$$\mathbf{s}_t = \tanh\left(\mathbf{W}\left[\mathbf{h}_t : \mathbf{c}_t\right]\right), \quad (3)$$

where $\mathbf{c}_t$ is the context vector produced from the encoder hidden states and $\mathbf{W}$ is a parameter matrix.

Here, we calculate the probability of action a_t according to the type of its frontier field:

- *Composite*. We adopt an APPLYCONSTR action to expand the field or a REDUCE action to complete the field.[3] The probability of using APPLYCONSTR[c] is defined as follows:

$$\begin{aligned} &p\left(a_t = \text{APPLYCONSTR}[c] | a_{<t}, \boldsymbol{x}\right) \\ &= \text{softmax}\left(\mathbf{E}(c)^\top \mathbf{W} \mathbf{s}_t\right) \end{aligned} \quad (4)$$

  where $\mathbf{E}(c)$ denotes the embedding of the constructor c.
- *Primitive*. We apply a GENTOKEN action to produce a token v, which is either generated from the vocabulary or copied from the input NL description. Formally, the probability of using GENTOKEN[v] can be decomposed into two parts:

$$\begin{aligned} &p\left(a_t = \text{GENTOKEN}[v] | a_{<t}, \boldsymbol{x}\right) \\ &= p\left(\text{gen} | a_{<t}, \boldsymbol{x}\right) p_{gen}\left(v | a_{<t}, \boldsymbol{x}\right) + \\ &\quad \left(1 - p\left(\text{gen} | a_{<t}, \boldsymbol{x}\right)\right) p_{copy}\left(v | a_{<t}, \boldsymbol{x}\right), \end{aligned} \quad (5)$$

  where $p\left(\text{gen} | a_{<t}, \boldsymbol{x}\right)$ is modeled as $\text{sigmoid}\left(\mathbf{W}\mathbf{s}_t\right)$.

Please note that our proposed dynamic selection of branch expansion orders does not affect other aspects of the model.

3 Dynamic Selection of Branch Expansion Orders

In this section, we extend the conventional Seq2Tree model with a context-based *branch selector*, which dynamically determines optimal expansion orders of branches for multi-branch AST nodes. In the following subsections, we first illustrate the elaborately-designed branch selector module and then introduce how to train the extended Seq2Tree model via reinforcement learning in detail.

[2] We also note that the field with sequential cardinality will be expanded to multiple branches. However, in this work, we do not consider this scenario, which is left as future work.

[3] REDUCE action can be considered as a special APPLYCONSTR action

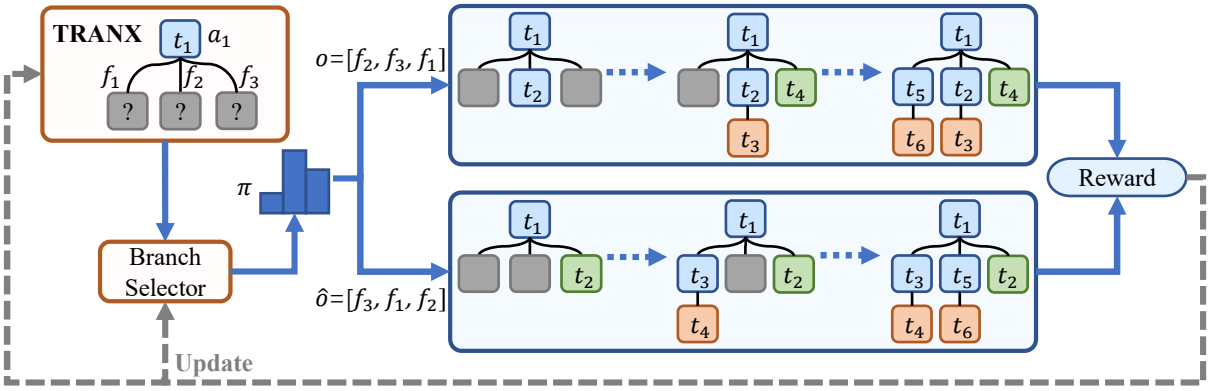


Figure 2: The reinforced training of the extended TRANX model with branch selector. We first fed the information of field and parent node into branch selector. Then, from the policy probability distribution of branch selector, we sample an order o and infer an order $\hat{o}$. Finally, we calculate the reward based on the model loss difference between o and $\hat{o}$, and use the gradients to update parameters of the extended model.

3.1 Branch Selector

As described in Section 2.2, the action prediction at each timestep is mainly affected by its previous action, frontier field and the action of its parent node. Thus, it is reasonable to construct the branch selector determining optimal expansion orders of branches according to these three kinds of information.

Specifically, given a multi-branch node n_t at timestep t, where the ASDL grammar of action a_t contains m fields $[f_1, f_2, ...f_m]$, we feed the branch selector with three vectors: 1) $\mathbf{E}(f_i)$: the embedding of field f_i, 2) $\mathbf{E}(a_t)$: the embedding of action a_t, 3) $\mathbf{s}_t$: the decoder hidden state, and then calculate the priority score of expanding fields as follows:

$$Score(f_i) = \mathbf{W}_2(\tanh(\mathbf{W}_1[\mathbf{s}_t : \mathbf{E}(a_t) : \mathbf{E}(f_i)])), \quad (6)$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_1 \times d_2}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_2 \times 1}$ are learnable parameters.[4]

Afterwards, we normalize priority scores of expanding all fields into a probability distribution:

$$p_{n_t} = \text{softmax}([Score(f_1) : \cdots : Score(f_m)]). \quad (7)$$

Based on the above probability distribution, we can sample m times to form a branch expansion order $o = [f_{o_1}, ..., f_{o_m}]$, of which the policy probability is computed as

$$\pi(o) = \prod_{i=1}^{m} p_{n_t}(f_{o_i} | f_{o_{<i}}). \quad (8)$$

[4]We omit the bias term for clarity.

It is notable that during the sampling of f_{o_i}, we mask previously sampled fields $f_{o_{<i}}$ to ensure that duplicate fields will not be sampled.

3.2 Training with Reinforcement Learning

During the generation of ASTs, with the above context-based branch selector, we deal with multi-branch nodes according to the dynamically determined order instead of the standard left-to-right order. However, the non-differentiability of multi-step expansion order selection and how to determine the optimal expansion order lead to challenges for the model training. To deal with these issues, we introduce reinforcement learning to train the extended Seq2Tree model in an end-to-end way.

Concretely, we first pre-train a conventional Seq2Tree model. Then, we employ self-critical training with a reward function that measures loss difference between different branch expansion orders to train the extended Seq2Tree model.

3.2.1 Pre-training

It is known that a well-initialized network is very important for applying reinforcement learning (Kang et al., 2020). In this work, we require the model to automatically quantify effects of different branch expansion orders on the quality of the generated action sequences. Therefore, we expect that the model has the basic ability to generate action sequences in random order at the beginning. To do this, instead of using the pre-order traversal based action sequences, we use the randomly-organized action sequences to pre-train the Seq2Tree model.

Concretely, for each multi-branch node in an AST, we sample a branch expansion order from a

uniform distribution, and then reorganize the corresponding actions according to the sampled order. We conduct the same operations to all multi-branch nodes of the AST, forming a new training instance. Finally, we use the regenerated training instances to pre-train our model.

In this way, the pre-trained Seq2Tree model acquires the preliminary capability to make predictions in any order.

3.2.2 Self-Critical Training

With the above initialized parameters, we then perform self-critical training (Rennie et al., 2017; Kang et al., 2020) to update the Seq2Tree model with branch selector.

Specifically, we train the extended Seq2Tree model by combining the MLE objective and RL objective together. Formally, given the training instance $(\boldsymbol{x}, \boldsymbol{a})$, we first apply the sampling method described in section 3.1 to all multi-branch nodes, reorganizing the initial action sequence $\boldsymbol{a}$ to form a new action sequence $\boldsymbol{a_o}$, and then define the model training objective as

$$\mathcal{L} = \mathcal{L}_{mle}(\boldsymbol{a_o}|\boldsymbol{x};\theta) + \frac{\lambda}{|\mathbf{N}_{mb}|}\sum_{n\in\mathbf{N}_{mb}}\mathcal{L}_{rl}(o;\theta), \tag{9}$$

where $\mathcal{L}_{mle}(*)$ denotes the conventional training objective defined in Equation 1, $\mathcal{L}_{rl}(*)$ is the negative expected reward of branch expansion order o for the multi-branch node n, λ is a balancing hyper-parameter, $\mathbf{N}_{mb}$ denotes the set of multi-branch nodes in the training instance, and θ denotes the parameter set of our enhanced model.

More specifically, $\mathcal{L}_{rl}(*)$ is defined as

$$\begin{aligned}\mathcal{L}_{rl}(o;\theta) &= -\mathbb{E}_{o\sim\pi}[r(o)] \\ &\approx -r(o), o\sim\pi,\end{aligned} \tag{10}$$

where we approximate the expected reward with the loss of an order o sampled from the policy π.

Inspired by successful applications of self-critical training in previous studies (Rennie et al., 2017; Kang et al., 2020), we propose the reward $r(*)$ to accurately measure the effect of any order on the model performance. As shown in Figure 2, we calculate the reward using two expansion orders of branches: one is o sampled from the policy π, and the other is $\hat{o}$ inferred from the policy π with the maximal generation probability:

$$r(o) = (\mathcal{L}_{mle}(\hat{o}) - \mathcal{L}_{mle}(o)) * (\max(\eta - p(o), 0)). \tag{11}$$

Please note that we extend the standard reward function by setting a threshold η to clip the reward, which can prevent the network from being over-confident in current expansion order of branches.

Finally, we apply the REINFORCE algorithm (Williams, 1992) to compute the gradient:

$$\nabla_\theta \mathcal{L}_{rl} \approx -r(o)\,\nabla_\theta \log p_\theta(o). \tag{12}$$

4 Experiments

To investigate the effectiveness and generalizability of our model, we carry out experiments on several commonly-used datasets.

4.1 Datasets

Following previous studies (Yin and Neubig, 2018, 2019; Xu et al., 2020), we use the following four datasets:

- **DJANGO** (Oda et al., 2015). This dataset totally contains 18,805 lines of Python source code, which are extracted from the Django Web framework, and each line is paired with an NL description.
- **ATIS**. This dataset is a set of 5,410 inquiries of flight information, where the input of each example is an NL description and its corresponding output is a short piece of code in lambda calculus.
- **GEO**. It is a collection of 880 U.S. geographical questions, with meaning representations defined in lambda logical forms like ATIS.
- **CONALA** (Yin et al., 2018). It totally consists of 2,879 examples of manually annotated NL questions and their Python solutions on STACK OVERFLOW. Compared with DJANGO, the examples of CONALA cover real-world NL queries issued by programmers with diverse intents, and are significantly more difficult due to its broad coverage and high compositionality of target meaning representations.

4.2 Baselines

To facilitate the descriptions of experimental results, we refer to the enhanced TRANX model as **TRANX-RL**. In addition to TRANX, we compare our enhanced model with several competitive models:

- **TRANX (w/ pre-train)**. It is an enhanced version of TRANX with pre-training. We

Model	DJANGO Acc.	ATIS Acc.	GEO Acc.	CONALA BLEU / Acc.
COARSE2FINE (Dong and Lapata, 2018)†	–	87.7	88.2	–
TRANX (Yin and Neubig, 2019)†	77.3 ±0.4	87.6 ±0.1	88.8 ±1.0	24.35 ±0.4 / 2.5 ±0.7
TREEGEN (Sun et al., 2020)	–	88.1 ±0.6	–	–
TRANX	77.2 ±0.6	87.6 ±0.4	88.8 ±1.0	24.38 ±0.5 / 2.2 ±0.5
TRANX (w/ pre-train)	77.5 ±0.4	87.8 ±0.7	88.4±1.1	24.57 ±0.5 / 1.4 ±0.3
TRANX-R2L	75.9 ±0.8	87.5 ±0.9	86.4 ±1.0	24.88 ±0.5 / 2.4 ±0.5
TRANX-RAND	74.6 ±1.1	86.4 ±1.4	81.7 ±1.8	19.73 ±1.1 / 1.6 ±0.6
TRANX-RL (w/o pre-train)	76.3 ±0.7	87.2 ±0.8	87.1 ±1.6	23.38 ±0.8 / 2.1 ±0.2
TRANX-RL	**77.9** ±0.5	**89.1** ±0.5	**89.5** ±1.2	**25.47** ±0.7 / **2.6** ±0.4

Table 2: The performance of our model in comparison with various baselines. We report the mean performance and standard deviation over five random runs. † indicates the scores are previously reported ones. Note that we only report the result of TREEGEN on ATIS, since it is the only dataset with released code for preprocessing.

compare with it because our model involves a pre-training stage.

- **COARSE2FINE** (Dong and Lapata, 2018). This model adopts a two-stage decoding strategy to produce the action sequence. It first generates a rough sketch of its meaning, and then fills in missing detail.
- **TREEGEN** (Sun et al., 2020). It introduces the attention mechanism of Transformer (Vaswani et al., 2017), and a novel AST reader to incorporate grammar and AST structures into the network.
- **TRANX-R2L**. It is a variant of the conventional TRANX model, which deals with multi-branch AST nodes in a right-to-left manner.
- **TRANX-RAND**. It is also a variant of the conventional TRANX model dealing with multi-branch AST nodes in a random order.
- **TRANX-RL (w/o pre-train)**. In this variant of TRANX-RL, we train our model from scratch. By doing so, we can discuss the effect of pre-training on our model training.

To ensure fair comparisons, we use the same experimental setup as TRANX (Yin and Neubig, 2018). Concretely, the sizes of action embedding, field embedding and hidden states are set to 128, 128 and 256, respectively. For decoding, the beam sizes for GEO, ATIS, DJANGO and CONALA are 5, 5, 15 and 15, respectively. We pre-train models in 10 epochs for all datasets. we determine the λs as 1.0 according to the model performance on validation sets. As in previous studies (Alvarez-Melis and Jaakkola, 2017; Yin and Neubig, 2018, 2019), we use the exact matching accuracy (Acc) as the evaluation metric for all datasets. For CONALA, we use the corpus-level BLEU (Yin et al., 2018) as a complementary metric.

4.3 Main Results

Table 2 reports the main experimental results. Overall, our enhanced model outperforms baselines across all datasets. Moreover, we can draw the following conclusions:

First, our reimplemented TRANX model achieves comparable performance to previously reported results (Yin and Neubig, 2019) (TRANX). Therefore, we confirm that our reimplemented TRANX model are convincing.

Second, compared with TRANX, TRANX-R2L and TRANX-RAND, our TRANX-RL exhibits better performance. This result demonstrates the advantage of dynamically determining branch expansion orders on dealing with multi-branch AST nodes.

Third, the TRANX model with pre-training does not gain a better performance. In contrast, removing the model pre-training leads to the performance degradation of our TRANX-RL model. This result is consistent with the conclusion of previous studies (Wang et al., 2018; Kang et al., 2020) that the pre-training is very important for the applying reinforcement learning.

4.4 Effects of the Number of Multi-branch Nodes

As implemented in related studies on other NLG tasks, such as machine translation (Bahdanau et al., 2015), we individually split two relatively large

Model	DJANGO Acc.	ATIS Acc.	GEO Acc.	CONALA Acc.
TRANX	77.26±0.8	94.02±0.8	89.75±0.8	25.19±0.6
TRANX-R2L	76.88±1.0	93.80±0.3	89.28±1.1	24.74±0.7
TRANX-RL	**78.98**±0.9	**94.87**±0.5	**90.64**±0.9	**26.90**±0.6

Table 3: Performance of our model in predicting actions for child nodes of multi-branch nodes.

	TRANX	TRANX-R2L	TRANX-RL
0	88.37	**93.02**	90.11
1	**100**	**100**	**100**
2	**100**	**100**	**100**
3	78.94	81.57	**89.47**
4	**96.93**	**96.93**	**96.93**
5	**95.65**	95.23	**95.65**
≥6	78.75	75.00	**80.63**

Table 4: Accuracy on different data groups of ATIS according to the number of multi-branch nodes.

	TRANX	TRANX-R2L	TRANX-RL
0	**98.30**	91.52	97.67
1	**90.00**	**90.00**	**90.00**
2	85.50	84.70	**86.17**
3	66.66	63.60	**67.81**
4	54.16	48.33	**57.50**
5	**28.88**	26.66	**28.88**
≥6	**12.35**	**12.35**	**12.35**

Table 5: Accuracy on different data groups of DJANGO according to the number of multi-branch nodes.

datasets (DJANGO and ATIS) into different groups according to the number of multi-branch AST nodes, and report the performance of various models on these groups of datasets.

Tables 4 and 5 show the experimental results. On most groups, TRANX-RL achieves better or equal performance than other models. Therefore, we confirm that our model is general to datasets with different numbers of multi-branch nodes.

4.5 Accuracy of Action Predictions for the Child Nodes

Given a multi-branch node, its child nodes have an important influence in the subtree. Therefore, we focus on the accuracy of action predictions for the child nodes.

For fair comparison, we predict actions with previous ground-truth history actions as inputs. Table 3 reports the experimental results. We observe that TRANX-RL still achieves higher prediction accuracy than other baselines on most groups, which proves the effectiveness of our model again.

4.6 Case Study

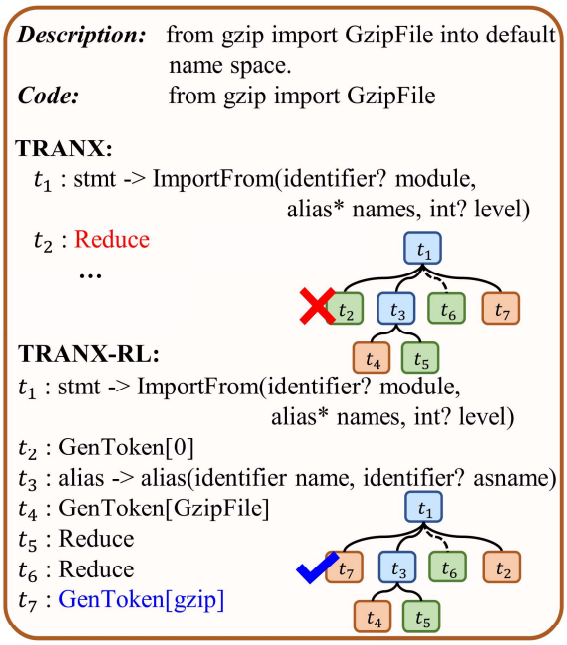
Figure 3 shows two examples from DJANGO. In the first example, TRANX first generates the leftmost child node at the timestep t_2, incorrectly predicting GENTOKEN['gzip'] as REDUCE action. By contrast, TRANX-RL puts this child node in the last position and successfully predict its action, since our model benefits from the previously generated token 'GzipFile' of the sibling node, which frequently occurs with 'gzip'.

In the second example, TRANX incorrectly predicts the second child node at the t_{10}-th timestep, while TRANX-RL firstly predicts it at the timestep t_6. We think this error results from the sequentially generated nodes and the errors in early timesteps would accumulatively harm the predictions of later sibling nodes. By comparison, our model can flexibly generate subtrees with shorter lengths, alleviating error accumulation.

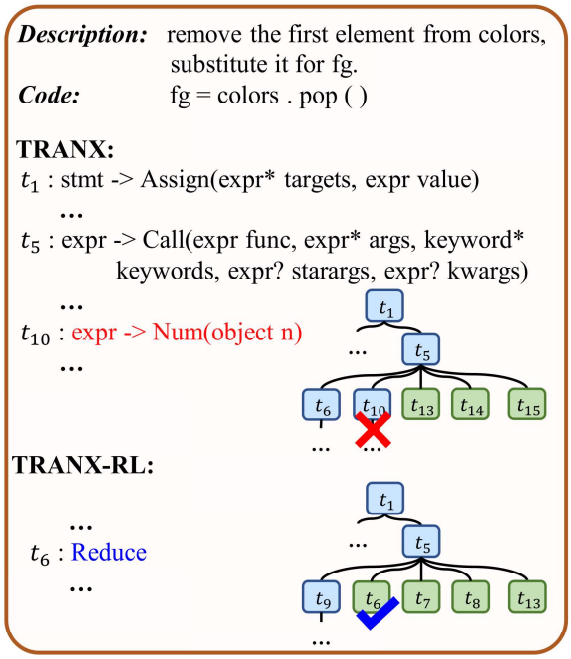
5 Related Work

With the prosperity of deep learning, researchers introduce neural networks into code generation. In this aspect, Ling et al. (2016) first explore a Seq2Seq model for code generation. Then, due to the advantage of tree structure, many attempts resort to Seq2Tree models, which represent codes as trees of meaning representations (Dong and Lapata, 2016; Alvarez-Melis and Jaakkola, 2017; Rabinovich et al., 2017; Yin and Neubig, 2017, 2018; Sun et al., 2019, 2020).

Typically, Yin and Neubig (2018) propose TRANX, which introduces ASTs as intermediate representations of codes and has become the most influential Seq2Tree model. Then, Sun et al. (2019, 2020) respectively explore CNN and Transformer



(a) The first example.



(b) The second example.

Figure 3: Two DJANGO examples produced by different models.

architectures to model code generation. Unlike these work, Shin et al. (2019) present a Seq2Tree model to generate program fragments or tokens interchangeably at each generation step. From another perspective, Xu et al. (2020) exploit external knowledge to enhance neural code generation model. Generally, all these Seq2Tree models generate ASTs in pre-order traversal, which, however, is not suitable to handle all multi-branch AST nodes. Different from the above studies that deal with multi-branch nodes in left-to-right order, our model determines the optimal expansion orders of branches for multi-branch nodes.

Some researchers have also noticed that the selection of decoding order has an important impact on the performance of neural code generation models. For example, Alvarez-Melis and Jaakkola (2017) introduce a doubly RNN model that combines width and depth recurrences to traverse each node. Dong and Lapata (2018) firstly generate a rough code sketch, and then fill in missing details by considering the input NL description and the sketch. Gu et al. (2019a) present an insertion-based Seq2Seq model that can flexibly generate a sequence in an arbitrary order. In general, these researches still deal with multi-branch AST nodes in a left-to-right manner. Thus, these models are theoretically compatible with our proposed branch selector.

Finally, it should be noted that have been many NLP studies on exploring other decoding methods to improve other NLG tasks (Zhang et al., 2018; Su et al., 2019; Zhang et al., 2019; Welleck et al., 2019; Stern et al., 2019; Gu et al., 2019a,b). However, to the best of our knowledge, our work is the first attempt to explore dynamic selection of branch expansion orders for tree-structured decoding.

6 Conclusion and Future Work

In this work, we first point out that the generation of domainant Seq2Tree models based on pre-order traversal is not optimal for handling all multi-branch nodes. Then we propose an extended Seq2Tree model equipped with a context-based branch selector, which is capable of dynamically determining optimal branch expansion orders for multi-branch nodes. Particularly, we adopt reinforcement learning to train the whole model with an elaborate reward that measures the model loss difference between different branch expansion orders. Extensive experiment results and in-depth analyses demonstrate the effectiveness and generality of our proposed model on several commonly-used datasets.

In the future, we will study how to extend our branch selector to deal with indefinite branches caused by sequential field.

Acknowledgments

The project was supported by National Key Research and Development Program of China (Grant No. 2020AAA0108004), National Natural Science Foundation of China (Grant No. 61672440), Natural Science Foundation of Fujian Province of China (Grant No. 2020J06001), Youth Innovation Fund of Xiamen (Grant No. 3502Z20206059), and the Fundamental Research Funds for the Central Universities (Grant No. ZK20720200077). We also thank the reviewers for their insightful comments.

References

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

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of ACL*, pages 33–43.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of ACL*, pages 6559–6569.

Jiatao Gu, Qi Liu, and Kyunghyun Cho. 2019a. Insertion-based decoding with automatically inferred generation order. *Trans. Assoc. Comput. Linguistics*, pages 661–676.

Jiatao Gu, Changhan Wang, and Junbo Zhao. 2019b. Levenshtein transformer. In *Proceedings of NIPS*, pages 11179–11189.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of EMNLP*, pages 925–930.

Xiaomian Kang, Yang Zhao, Jiajun Zhang, and Chengqing Zong. 2020. Dynamic context selection for document-level neural machine translation via reinforcement learning. In *Proceedings of EMNLP*, pages 2242–2254.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of ACL*, pages 599–609.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of ASE*, pages 574–584.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of ACL*, pages 1139–1149.

Steven J. Rennie, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel. 2017. Self-critical sequence training for image captioning. In *Proceedings of CVPR*, pages 1179–1195.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Proceedings of NIPS*, pages 10825–10835.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. In *Proceedings of ICML*, pages 5976–5985.

Jinsong Su, Xiangwen Zhang, Qian Lin, Yue Qin, Junfeng Yao, and Yang Liu. 2019. Exploiting reverse target-side contexts for neural machine translation via asynchronous bidirectional decoding. *Artificial Intelligence*, 277.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of AAAI*, pages 7055–7062.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of AAAI*, pages 8984–8991.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of NIPS*, pages 5998–6008.

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of ECCV*, pages 420–436.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Proceedings of NIPS*, pages 6563–6573.

Sean Welleck, Kianté Brantley, Hal Daumé III, and Kyunghyun Cho. 2019. Non-monotonic sequential text generation. In *Proceedings of ICML*, pages 6716–6726.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Binbin Xie, Xiang Li, Yubin Ge, Jianwei Cui, Junfeng Yao, Bin Wang, and Jinsong Su. 2021. Improving tree-structured decoder training for code generation via mutual learning. In *Proceedings of AAAI*.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of ACL*, pages 6045–6052.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of MSR*, pages 476–486.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of ACL*, pages 440–450.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of EMNLP*, pages 7–12.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of ACL*, pages 4553–4559.

Biao Zhang, Deyi Xiong, Jinsong Su, and Jiebo Luo. 2019. Future-aware knowledge distillation for neural machine translation. *IEEE ACM Trans. Audio Speech Lang. Process.*, pages 2278–2287.

Xiangwen Zhang, Jinsong Su, Yue Qin, Yang Liu, Rongrong Ji, and Hongji Wang. 2018. Asynchronous bidirectional decoding for neural machine translation. In *Proceedings of AAAI*, pages 5698–5705.