Improving Tree-Structured Decoder Training for Code Generation via Mutual Learning

Binbin Xie[1,2], **Jinsong Su**[1,2 *], **Yubin Ge**[3], **Xiang Li**[4], **Jianwei Cui**[4], **Junfeng Yao**[1] and **Bin Wang**[4]

1. Xiamen University, Xiamen, China
2. Peng Cheng Laboratory, Shenzhen, China
3. University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
4. Xiaomi AI Lab, Beijing, China

xdblb@stu.xmu.edu.cn, {jssu, yao0010}@xmu.edu.cn,
yubinge2@illinois.edu, {lixiang21, cuijianwei, wangbin11}@xiaomi.com,

Abstract

Code generation aims to automatically generate a piece of code given an input natural language utterance. Currently, among dominant models, it is treated as a sequence-to-tree task, where a decoder outputs a sequence of actions corresponding to the pre-order traversal of an Abstract Syntax Tree. However, such a decoder only exploits the preorder traversal based preceding actions, which are insufficient to ensure correct action predictions. In this paper, we first throughly analyze the context modeling difference between neural code generation models with different traversals based decodings (preorder traversal vs breadth-first traversal), and then propose to introduce a mutual learning framework to jointly train these models. Under this framework, we continuously enhance both two models via mutual distillation, which involves synchronous executions of two one-to-one knowledge transfers at each training step. More specifically, we alternately choose one model as the student and the other as its teacher, and require the student to fit the training data and the action prediction distributions of its teacher. By doing so, both models can fully absorb the knowledge from each other and thus could be improved simultaneously. Experimental results and in-depth analysis on several benchmark datasets demonstrate the effectiveness of our approach. We release our code at https://github.com/DeepLearnXMU/CGML.

Introduction

As an indispensable text generation task, code generation mainly focuses on automatically generating a snippet of code given a natural language (NL) utterance. Due to its great potential in facilitating software development and revolutionizing end-user programming, it has always been one of hot research topics in the communities of natural language processing and software engineering.

To achieve this goal, previous studies for code generation either require manually-designed grammar and lexicons (Zettlemoyer 2009; Zettlemoyer and Collins 2007) or exploit features for candidate logical forms ranking (Liang, Jordan, and Klein 2011). With the prosperity of deep learning in natural language processing, dominant models have now evolved into neural models, which possess the powerful capacity of feature learning and representation. Among them, the most prevalent one is the sequence-to-tree (Seq2Tree) models (Dong and Lapata 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Dong and Lapata 2018; Shin et al. 2019; Sun et al. 2020), all of which are based on an encoder-decoder framework. Specifically, an encoder learns the word-level semantic representations of an input NL utterance, and then a tree-structured decoder outputs a sequence of actions, which corresponds to the preorder traversal of an Abstract Syntax Tree (AST) and can be further converted into the code, 1) the structure of AST can be used to shrink the search space, ensuring the generation of well-formed code; and 2) the structural information of AST helps to model the information flow within the neural network, which naturally reflects the recursive structure of programming languages.

Despite the success of the above models, limited by the decoding manner based on the pre-order traversal, there still exists a serious defect in their decoders. Specifically, at each timestep, the current action can be only predicted from the vertical preceding actions. Since the dependencies of actions may come from other directions, i.e. the actions in horizontal direction, it may be insufficient to accurately predict the current action only with preceding actions in vertical direction. This happens especially when an important action locates in horizontal direction. Another possible case would be the strength of the dependencies on preceding actions turn to be weak with the increasing distances to those actions. Moreover, prediction errors in preceding actions sometimes even hurt the prediction of current action. Therefore, it is worth exploring the context in other direction to complement the currect context of the pre-order traversal based decoder.

In this paper, we first explore the Seq2Tree model with breadth-first traversal based decoding, and then analyze the context modeling difference between different traversals based decodings (pre-order traversal vs. breath-first traversal). More importantly, we propose to introduce a mutual learning framework (Zhang et al. 2018b) to jointly train two Seq2Tree models with different traversals based de-

[*]Corresponding author.

codings, both of which are expected to benefit each other. Our motivation stems from the fact that these two models are able to capture the context from different directions. That is, the conventional Seq2Tree model focuses on the pre-order traversal based preceding actions, while the other one mainly pays attention to the preceding actions based on breadth-first traversal. Hence, such difference brings the potential in enhancing both models.

Under the proposed framework, we continuously improve both models via mutual distillation, which involves synchronous executions of two one-to-one transfers at each training step. More specifically, we alternately choose one model as the student and the other as its teacher, and require the student to fit the training data and match action prediction distributions of its teacher. By doing so, both models can fully absorb the knowledge from each other and thus could be improved simultaneously.

The contributions of our work are summarized as follows:

- We explore the neural code generation model with breadth-first traversal based decoding, and point out that models with different traversals can be complementary to each other in context modeling.
- We propose to introduce a mutual learning based model training framework for code generation, where Seq2Tree models with different traversals based decodings can continuously enhance each other.
- On several commonly-used datasets, we demonstrate the effectiveness and generality of our framework.

Background

In this work, we choose TRANX (Yin and Neubig 2018) as our basic model, which has been widely used due to its competitive performance (Yin and Neubig 2019; Shin et al. 2019; Xu et al. 2020). Note that both our explored decoder with breadth-first traversal and the mutual learning based model training framework are also applicable to other Seq2Tree models. In the following subsections, we first introduce how to convert NLs to code using TRANX, and then give a detailed description of the architecture of TRANX.

Converting NLs to Code Using TRANX

Typically, TRANX introduces ASTs as the intermediate meaning representations (MRs). Figure 1 gives an example of the NL utterance-to-code conversion for the Python code "*if six.PY3: pass*". Concretely, such a conversion involves two stages: (1) the transition from the NL utterance into an AST. During this process, Tranx outputs a sequence of ASDL formalism based actions, corresponding to the AST in pre-order traversal; and (2) A user specified function AST_to_MR(*) is called to convert the generated AST into the code.

Specifically, at each timestep, one of three types of actions is evoked to expand the non-terminal node of the partial AST:

(1) APPLYCONSTR[c] actions apply a constructor c to the opening composite frontier field which has the same type as c, populating the opening node using the fields in c. If the frontier field has sequential cardinality, the action appends the constructor to the list of constructors held by the field. Back to Figure 1, "n_1:APPLYCONSTR[stmt $\rightarrow$ If(expr test, stmt* body, stmt* orelse)]" acts on the root of the AST and has three fields to be expended.

(2) REDUCE actions mark the completion of the generation of child values for a field with optional (?) or multiple ($*$) cardinalities. In Figure 1, "n_7:REDUCE" means the completion of the field "stmt* body" of "n_1:APPLYCONSTR [stmt $\rightarrow$ If(expr test, stmt* body, stmt* orelse)]".

(3) GENTOKEN[v] actions populate a (empty) primitive frontier field with a token v. Specifically, the field of node with primitive types should be filled with GENTOKEN[v] action. For instance, "n_4:GENTOKEN[six]" means the token "six" fills the identifier field of "APPLYCONSTR[expr $\rightarrow$ Name(identifier id)]".

TRANX

TRANX is based on an attentional encoder-decoder framework, where a BiLSTM encoder is used to learn word-level semantic representations $\{h_i\}_{i=1}^{|X|}$ of the input NL utterance X, converted by an LSTM decoder into a sequence of tree-constructing actions.

Specifically, during the timestep t of decoding, its LSTM cell reads the embedding $e(a_{t-1})$ of the previous action a_{t-1}, the temporary hidden state $\tilde{s}_{t-1}$, the vector p_t concatenated by the hidden state of the parent AST node and the type embedding of the current AST node, and the previous hidden state s_{t-1} to generate the hidden state vectors s_t and $\tilde{s}_t$ as:

$$s_t = f_{\text{LSTM}}([e(a_{t-1}); \tilde{s}_{t-1}; p_t], s_{t-1}), \tag{1}$$

$$\tilde{s}_t = \tanh(W_s[c_t; s_t]), \tag{2}$$

where W_s is a parameter matrix and the attentional context vector c_t is produced from $\{h_i\}$.

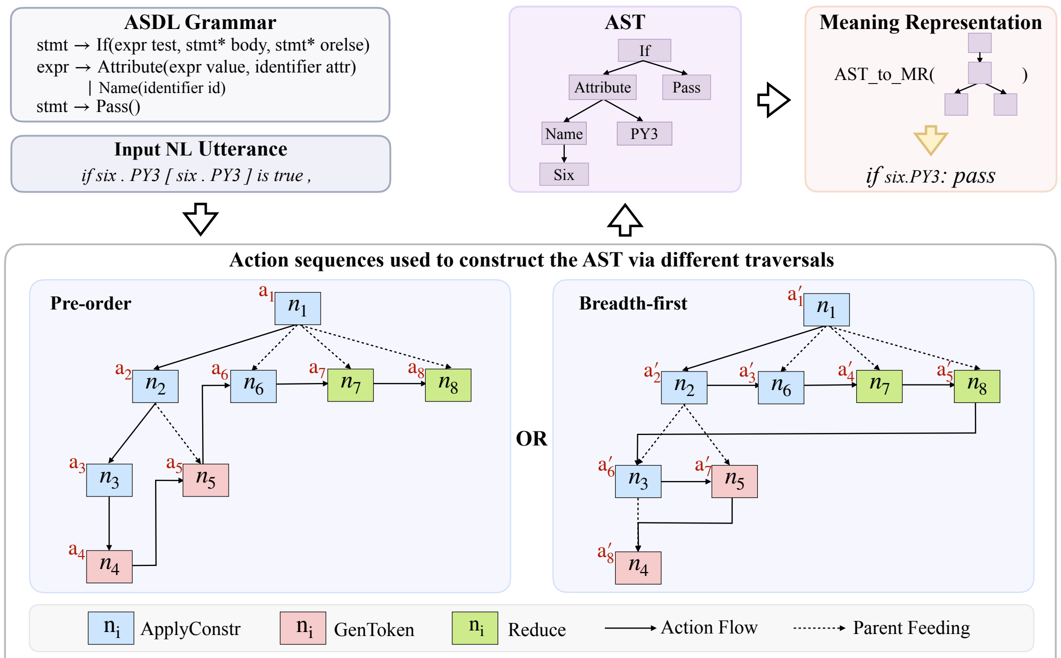
Finally, the probability of applying the action a_t is computed according to the type of the current node specified by the corresponding action of its parent node:

(1) ***Primitive* type.** In this case, TRANX takes an action a_t=GENTOKEN[v], where the token v can be either generated directly or copied from the input NL utterance as follows:

$$\begin{aligned} & p(a_t = \text{GENTOKEN}[v]|a_{<t}, \mathbf{x}) \\ = & \, p(gen|a_{<t}, \mathbf{x}) \cdot p_{gen}(v|a_{<t}, \mathbf{x}) \\ + & \, (1 - p(gen|a_{<t}, \mathbf{x})) \cdot p_{copy}(v|a_{<t}, \mathbf{x}). \end{aligned} \tag{3}$$

Here, we use three softmax functions based on $\tilde{s}_t$ to the probability $p(gen|a_{<t}, \mathbf{x})$ of choosing the generation operation, the probability $p_{gen}(v|a_{<t}, \mathbf{x})$ of generating v and the probability $p_{copy}(v = x_i|a_{<t}, \mathbf{x})$ of selecting to copy x_i, respectively.

(2) ***Composite* type.** On this condition, TRANX either applies an APPLYCONSTR action to produce the current node, or employs a REDUCE operation to terminate some node expansion from the AST parent node. Formally, the probability $p(a_t|a_{<t}, \mathbf{x})$ of the action a_t is computed as softmax$(e(a_t)^{\mathsf{T}} W_a \tilde{s}_t)$, where W_a is the parameter matrix of the linear mapping.



Node	Action		Action Definition
	Pre-order	Breadth-first	
n_1	a_1	a'_1	ApplyConstr[stmt → If(expr test, stmt * body, stmt * orelse)]
n_2	a_2	a'_2	ApplyConstr[expr → Attribute(expr value, identifier attr)]
n_3	a_3	a'_6	ApplyConstr[expr → Name(identifier id)]
n_4	a_4	a'_8	GenToken[six]
n_5	a_5	a'_7	GenToken[PY3]
n_6	a_6	a'_3	ApplyConstr[stmt → Pass()]
n_7	a_7	a'_4	Reduce
n_8	a_8	a'_5	Reduce

Figure 1: The procedure of converting an input NL utterance into a snippet of code.

Training Objective. Given a training corpus $D = \{(\mathbf{x}, \mathbf{a})\}$, the training objective $J_{\text{MLE}}(D;\theta)$ is defined as

$$J_{\text{MLE}}(D;\theta) = \sum_{(\mathbf{x},\mathbf{a})\in D} J_{\text{MLE}}(\mathbf{x},\mathbf{a};\theta), \quad (4)$$

$$J_{\text{MLE}}(\mathbf{x},\mathbf{a};\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log p(a_t|a_{<t},\mathbf{x};\theta). \quad (5)$$

Our Approach

In this section, we first briefly describe two Seq2Tree models with different traversals (pre-order traversal and breadth-first traversal) based decodings. Then, we propose to introduce mutual learning (Zhang et al. 2018b) based model training framework into code generation, where these two models can be enhanced simultaneously.

Seq2Tree Models

TRANX. We omit the description of TRANX, which has been provided in Background section.

TRANX$'$. As a variant of TRANX, TRANX$'$ provides a new perspective of code generation, where the AST is generated via breadth-first traversal and thus the AST context in horizontal direction can be leveraged for action predictions.

To train TRANX$'$, we traverse the AST of each training example in breadth-first order to generate an reference action sequence $\mathbf{a}'$. As shown in the middle of Figure 1, for the code "*if six.PY3 : pass*", we traverse its AST in the breadth-first order and then reorganize the corresponding actions into an action sequence: $\mathbf{a}'$=a$'_1$, a$'_2$,..., a$'_8$. Note that although reference action sequences of TRANX and TRANX$'$ are different, they are from the same AST.

Likewise, we then parameterize the prediction probability using a LSTM netowrk: $p(\mathbf{a}'|\mathbf{x}) = \prod_{t'=1}^{T} p(a'_{t'}|a'_{<t'},\mathbf{x})$. The main equations of this LSTM are almost the same as those of TRANX, the only difference is that the definitions of preceding actions are different. Very importantly, $e(a_{t-1})$ of Equation 1 is replaced with $e(a'_{t'-1})$.

Apparently, during different traversals based decodings, TRANX and TRANX$'$ focus on the AST-based context from different directions. For example, for the action prediction

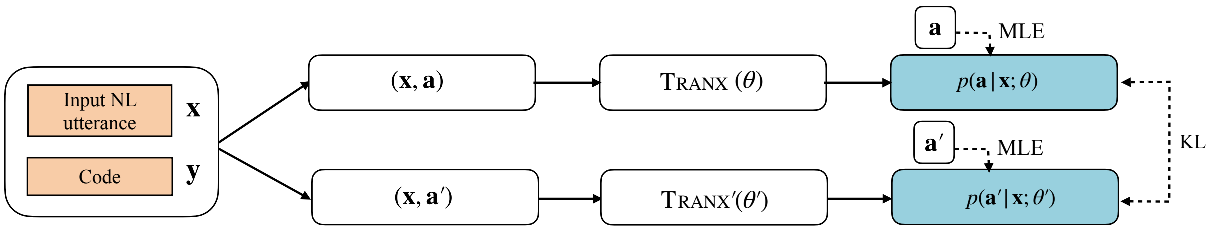


Figure 2: Mutual learning based model training framework. Particularly, TRANX and TRANX′ share the same encoder.

Algorithm 1 The training procedure of our framework.

```
Input: Training set $D$, validation set $D_v$
 1: repeat
 2:     repeat
 3:         Load a batch size of instances $B \in D$
 4:         $J(B;\theta) = 0$
 5:         $J(B;\theta') = 0$
 6:         for each instance $(\mathbf{x}, \mathbf{a}) \in B$ do
 7:             Traverse the AST of $\mathbf{a}$ in a breadth-first order to form a reference action sequence $\mathbf{a}'$ for TRANX′
 8:             Calculate the loss $J(\mathbf{x}, \mathbf{a}; \theta)$ of TRANX
 9:             Calculate the loss $J(\mathbf{x}, \mathbf{a}'; \theta')$ of TRANX′
10:             $J(B;\theta)$ += $J(\mathbf{x}, \mathbf{a}; \theta)$
11:             $J(B;\theta')$ += $J(\mathbf{x}, \mathbf{a}'; \theta')$
12:         end for
13:         Minimize $J(B, \theta)$ to update $\theta$
14:         Minimize $J(B, \theta')$ to update $\theta'$
15:         Save the best models according to their performance on $D_v$
16:     until no more batches
17: until both $\theta$ and $\theta'$ converge
```

of the considered node n_6, TRANX exploits the context encoded by nodes n_1 and n_5 while TRANX′ makes use of the context of its sibling n_2. Intuitively, both nodes n_2 and n_1 are closely-related nodes of n_6, possessing important impact on the action prediction of n_6. Therefore, we believe that TRANX and TRANX′ can complement each other.

Mutual Learning Based Model Training

As illustrated in Figure 2, we then propose to introduce a mutual learning framework to jointly train TRANX and TRANX′. Need to add that our framework is also suitable for joint training of more than two models.

In order to facilitate the understanding of our framework, we also depict its training procedure in Algorithm 1. Using this framework, we continuously enhance these two models via mutual distillation, where two one-to-one knowledge transfers in reverse directions are synchronously executed at each training step (Lines 6-15). During the one-to-one transfer procedure, we expect each model to fully learn knowledge from not only its own training data but also the other.

To this end, we require each considered model to fit the training data and match the action prediction distributions of the other model. In addition to the conventional MLE loss on training data, we introduce one Kullback-Leibler (KL) loss to quantify the action prediction distribution divergence between the considered model and the other one. Suppose θ and θ' to be the parameter sets of TRANX and TRANX′, we define the following objective function to update their parameters:

$$J(D,\theta) = \sum_{(\mathbf{x},\mathbf{a})\in D} \{J_{\text{MLE}}(\mathbf{x},\mathbf{a};\theta) + \tag{6}$$

$$\frac{\lambda}{T} \cdot \sum_{n\in\mathbf{z}} \text{KL}(p(a'_{t'(n)}|a'_{<t'(n)},\mathbf{x};\theta')||p(a_{t(n)}|a_{<t(n)},\mathbf{x};\theta))\},$$

$$J(D,\theta') = \sum_{(\mathbf{x},\mathbf{a}')\in D} \{J_{\text{MLE}}(\mathbf{x},\mathbf{a}';\theta') + \tag{7}$$

$$\frac{\lambda}{T} \cdot \sum_{n\in\mathbf{z}} \text{KL}(p(a_{t(n)}|a_{<t(n)},\mathbf{x};\theta)||p(a'_{t'(n)}|a'_{<t'(n)},\mathbf{x};\theta'))\},$$

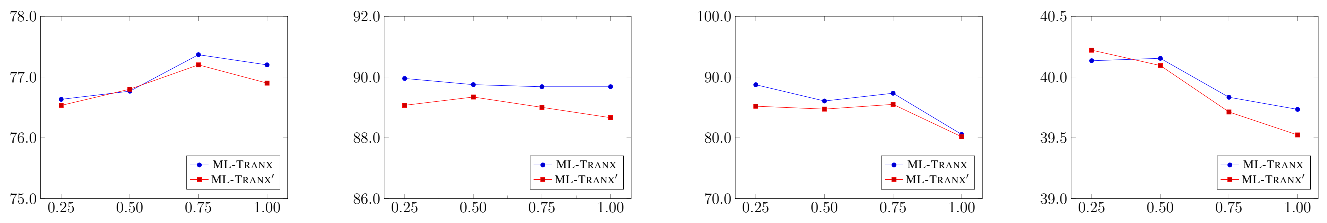
where $t(n)$ and $t'(n)$ denote the timesteps of the AST node n during the pre-order and breadth-first traversals of the AST $\mathbf{z}$, respectively, KL$(\cdot||\cdot)$ is the KL divergence, and λ is the coefficient used to control the impacts of different losses. Note that both our KL terms act on the AST node-level outputs of different models, that is, $p(a'_{t'(n)}|a'_{<t'(n)},\mathbf{x};\theta')$ and $p(a_{t(n)}|a_{<t(n)},\mathbf{x};\theta)$, where the AST node n may correspond to different timesteps of our two models. Back to Figure 1, the node n_6 corresponds to the 6-th and 3-rd timesteps of pre-order and breadth-first traversals, respectively.

We repeat the above-mentioned knowledge transfer process, until both models converge.

Experiments

Datasets

We carry out experiments on the following datasets: 1) **DJANGO** (Oda et al. 2015). This dataset consists of 18,805 lines of Python source code, with each line paired with an NL utterance. We split the dataset into training/validation/test sets containing 16,000/1,000/1,805 instances, respectively. 2) **ATIS**. This dataset includes NL questions of a flight database, with each question is annotated with a lambda calculus query. Following previous studies (Yin and Neubig 2018, 2019; Xu et al. 2020), we use the standard splits of training/validation/test sets, which contains 4,473/491/448 instances, respectively. 3) **GEO**. It contains NL questions about US geography paired with corresponding Prolog database queries. we use the standard splits of 600/280 training/test instances. 4) **IFTTT** (Quirk, Mooney, and Galley 2015). It consists of if-this-then-that programs, paired with NL utterances of their purpose. The dataset is split into 68,083 training, 5,171 validation and 3,868 test instances.



(a) **DJANGO** (b) **ATIS** (c) **GEO** (d) **IFTTT**

Figure 3: Effects of λ on the validation sets.

Baseline Models

We apply mutual learning to simultaneously train TRANX and TRANX′, which are respectively referred to as ML-TRANX and ML-TRANX′, avoiding the description confusion. In addition to the conventional TRANX and TRANX′, and some commonly-used baselines (See Table 1), we compare the enhanced models with the following models:

- **Ensemble** (Sennrich et al. 2017). Using this method, we first employ depth-first and bread-first model independently to obtain two k-best lists, and then re-score the combination of these two lists from these two models.
- **MTL-Tranx** and **MTL-Tranx′**. We apply multitask learning to jointly train TRANX and TRANX′ sharing the same encoder.
- **KD-TRANX**. We employ KD to transfer the knowledge of the TRANX′ with fixed parameters to enhance TRANX.
- **KD-TRANX′**. A TRANX′ enhanced by a fixed TRANX via KD.
- **ML2-TRANX**. Under our framework, two TRANX models with different initializations are jointly trained.
- **ML2-TRANX′**. Two TRANX′ models with different initializations are jointly trained under our framework.

Particularly, we report the average performance of two involved models for ML2-TRANX and ML2-TRANX′. We use the same experimental setup as (Yin and Neubig 2017). Specifically, we use 256 hidden units and 128-dimensional word vectors for NL utterance encoding, and tune the dimension of various embeddings on validation datasets for each corpus. We initialize all parameters by uniformly sampling within the interval [-0.1, 0.1]. Besides, we set the batch size as 10 and employ dropout after each layer, where the drop rate is sequentially set to 0.5, 0.3, 0.4, and 0.3 for our four datasets, respectively. To alleviate the instability of the model training, we run each model five times and report the average performance in terms of exact matching accuracy. Following the evaluation protocol in (Beltagy and Quirk 2016) for accuracies, we evaluate model performance on IFTTT at both channel and full parse tree (channel + function) levels.

Effect of λ

As shown in Equations 6 and 7, the coefficient λ is an important hyper-parameter that controls relative impacts of the MLE loss and the divergence losses. Thus, we first investigate the effect of λ on our proposed framework. To this end, we gradually vary λ from 0 to 1 with an increment of 0.25 at each step, and report the performance of our models using different λs on the validation datasets. Since there exists no validation set in GEO, we temporarily split the training data into two parts: 480 instances for training and 120 instances for validation, and use them to determine the optimal λ. However, with the optimal λ, we still use the whole training data in subsequent experiments.

Figure 3 shows the experimental results. According to the average performance of our models on four validation sets, we set λ as 0.75, 0.5, 0.25 and 0.25 for our four datasets in all experiments thereafter.

Main Results

Table 1 reports the overall experimental results. Both models involved are improved under our framework. Besides, we can draw the following conclusions:

(1) Our implemented TRANX achieves comparable performance to (Yin and Neubig 2019). Thus, we confirm that our reimplemented baseline is competitive.

(2) Both ML-TRANX and ML-TRANX′ significantly surpass their respective important baselines: TRANX and TRANX′. Besides, both of our enhanced models surpass DRNN. These results strongly validate the effectiveness of our proposed training framework.

(3) Both ML-TRANX and ML-TRANX′ outperforms Ensemble. This result is reasonable since TRANX and TRANX′ can interact with each other via mutual learning while they are independently trained when using ensemble.

(4) Compared with multitask learning based models: MTL-TRANX and MTL-TRANX′, both ML-TRANX and ML-TRANX′ achieve better performance. This is due to the fact that in addition to sharing encoder parameters, mutual learning can refine model training by working as regularization terms, which has also been mentioned in (Romero et al. 2015; Clark et al. 2019).

(5) ML-TRANX and ML-TRANX′ respectively show better performance than KD-TRANX and KD-TRANX′, verifying the necessity of multiple iterations of knowledge transfer.

(6) ML-TRANX and ML-TRANX′ perform better in different datasets. The underlying reason is that each instance contains only three layers in IFTTT, where the effectiveness of pre-order decoding is greatly restricted.

Model	DJANGO	ATIS	GEO	IFTTT
	ACC.	ACC.	ACC.	ACC. CHANNEL / FULL TREE
DRNN (Alvarez-Melis and Jaakkola 2017)	–	–	–	90.1 / 78.2
ASN (Rabinovich, Stern, and Klein 2017)	–	85.9	87.1	– / –
SEQ2AST-YN17 (Yin and Neubig 2017)	75.8	–	–	90.0 / 82.0
COARSE2FINE (Dong and Lapata 2018)	74.1	87.7	88.2	– / –
TRANX (Yin and Neubig 2019)	$77.3_{\pm0.4}$	$87.6_{\pm0.1}$	$88.8_{\pm1.0}$	– / –
TREECONV (Sun et al. 2019)	–	85.0	–	– / –
TREEGEN (Sun et al. 2020)	–	89.1	**89.6**	– / –
ENSEMBLE (Sennrich et al. 2017)	$78.6_{\pm0.4}$	$88.3_{\pm0.5}$	$88.8_{\pm0.7}$	$91.1_{\pm1.0}$ / $84.5_{\pm0.5}$
TRANX	$77.3_{\pm0.4}$	$87.7_{\pm0.5}$	$88.7_{\pm0.7}$	$91.0_{\pm1.0}$ / $82.8_{\pm0.5}$
MTL-TRANX	$78.2_{\pm0.1}$	$88.0_{\pm0.3}$	$89.0_{\pm0.8}$	$91.6_{\pm0.2}$ / $84.3_{\pm1.4}$
KD-TRANX	$78.1_{\pm0.3}$	$87.9_{\pm0.3}$	$88.8_{\pm0.6}$	$91.2_{\pm0.5}$ / $83.8_{\pm0.5}$
ML2-TRANX	$79.0_{\pm0.1}$	$87.8_{\pm1.1}$	$88.9_{\pm0.6}$	$91.5_{\pm0.4}$ / $84.9_{\pm0.1}$
ML-TRANX	**79.6**$_{\pm0.3}$	**89.3**$_{\pm0.3}$	$89.2_{\pm0.6}$	**92.0**$_{\pm0.3}$ / $85.2_{\pm1.6}$
TRANX′	$76.8_{\pm0.2}$	$86.9_{\pm0.3}$	$87.0_{\pm1.3}$	$90.1_{\pm0.2}$ / $80.1_{\pm1.0}$
MTL-TRANX′	$78.0_{\pm0.1}$	$87.7_{\pm0.1}$	$88.6_{\pm0.5}$	$91.7_{\pm0.1}$ / $84.7_{\pm0.7}$
KD-TRANX′	$77.6_{\pm0.4}$	$87.1_{\pm0.4}$	$88.3_{\pm0.4}$	$90.8_{\pm0.3}$ / $83.4_{\pm0.8}$
ML2-TRANX′	$77.8_{\pm0.2}$	$87.0_{\pm1.3}$	$87.9_{\pm0.3}$	$91.0_{\pm0.3}$ / $83.8_{\pm1.0}$
ML-TRANX′	$78.6_{\pm0.1}$	$88.4_{\pm0.2}$	$88.9_{\pm0.4}$	$91.9_{\pm0.1}$ / **85.7**$_{\pm0.3}$

Table 1: Main experimental results. All results shown in the upper part are directly cited from their corresponding papers.

Model	DJANGO				
	[1, 10]	[11, 20]	[21, 30]	[31, 40]	[41, ∞)
ENSEMBLE	92.9	79.7	42.9	**10.9**	–
TRANX	93.0	76.7	45.7	5.4	–
MTL-TRANX	94.7	81.5	44.0	8.7	–
KD-TRANX	93.8	79.1	44.0	6.5	–
ML2-TRANX	93.8	80.9	44.3	6.5	–
ML-TRANX	**95.0**	**82.2**	**48.6**	8.7	–
TRANX′	92.4	79.1	42.3	**10.9**	–
MTL-TRANX′	94.8	81.1	44.6	6.5	–
KD-TRANX′	93.8	80.9	43.4	8.7	–
ML2-TRANX′	93.8	80.0	46.0	**10.9**	–
ML-TRANX′	94.0	81.8	43.4	**10.9**	–

Table 2: Accuracy on different groups of DJANGO. Please note that on the group [41, ∞), none of the models performed well.

(7) Compared with ML2-TRANX, our TRANX still exhibits better performance. Likewise, TRANX performs better than ML2-TRANX′. Hence, we confirm that TRANX and TRANX′ are indeed able to benefit each other.

Performance by the Size of AST

Further, in order to inspect the generality of our proposed framework, we follow Yin and Neubig (2017) to split datasets into different groups according to their AST sizes, and then compare the model performance at each group.

Tables 2, 3 and 4 display the results on DJANGO, ATIS and GEO, respectively. [1]

[1]We do not display the results on IFTTT, since all of its instances have the same AST size.

Model	ATIS				
	[1, 10]	[11, 20]	[21, 30]	[31, 40]	[41, ∞)
ENSEMBLE	**93.3**	78.9	**96.8**	93.6	60.9
TRANX	88.3	84.2	95.3	92.8	60.9
MTL-TRANX	**93.3**	89.5	94.5	92.8	65.2
KD-TRANX	90.8	84.2	95.3	93.6	60.9
ML2-TRANX	90.0	88.0	96.0	93.6	60.9
ML-TRANX	**93.3**	**92.1**	**96.8**	93.6	**63.8**
TRANX′	85.0	84.2	96.0	92.8	56.5
MTL-TRANX′	90.0	89.5	95.3	**94.1**	56.5
KD-TRANX′	90.0	88.2	96.0	92.8	60.1
ML2-TRANX′	89.2	88.2	96.0	92.2	60.1
ML-TRANX′	91.7	89.5	96.0	93.6	60.9

Table 3: Accuracy on different groups of ATIS.

Model	GEO				
	[1, 10]	[11, 20]	[21, 30]	[31, 40]	[41, ∞)
ENSEMBLE	**98.1**	**93.8**	90.1	77.1	45.5
TRANX	**98.1**	**93.8**	88.7	72.2	45.5
MTL-TRANX	96.3	93.8	92.9	80.6	**54.5**
KD-TRANX	97.2	93.8	92.9	80.6	**54.5**
ML2-TRANX	**98.1**	**93.8**	92.9	80.6	45.5
ML-TRANX	**98.1**	92.6	**93.9**	**83.3**	**54.5**
TRANX′	96.3	92.6	88.8	72.2	45.5
MTL-TRANX′	96.3	92.6	89.8	80.6	36.4
KD-TRANX′	96.3	93.2	88.8	72.2	40.9
ML2-TRANX′	97.2	93.2	90.1	80.6	40.9
ML-TRANX′	**98.1**	**93.8**	91.8	80.6	45.5

Table 4: Accuracy on different groups of GEO.

We can observe that our enhanced models outperform or achieve comparable performance than their corresponding

Source	call the method line.lstrip [line . lstrip] , if the result starts with TRANSLATOR_COMMENT_MARK ,
TRANX	if not line.lstrip(TRANSLATOR_COMMENT_MARK): pass ✗
TRANX′	if line.lstrip(TRANSLATOR_COMMENT_MARK).startswith(TRANSLATOR_COMMENT_MARK): pass ✗
ENSEMBLE	if line.startswith(TRANSLATOR_COMMENT_MARK): pass ✗
MTL-TRANX	if line.lstrip().starts(TRANSLATOR_COMMENT_MARK):pass ✗
KD-TRANX	if line.startswith(TRANSLATOR_COMMENT_MARK): pass ✗
ML2-TRANX	if not line.startswith(TRANSLATOR_COMMENT_MARK): pass ✗
ML-TRANX	if line.lstrip().startswith(TRANSLATOR_COMMENT_MARK): pass ✓

Table 5: Codes produced by different models. Incorrect codes are marked in red while counterparts are marked in blue.

baselines respectively on almost all groups of datasets. Thus, we confirm again the effectiveness and generality of our proposed framework.

Case Study

We compare the 1-best codes produced by different code generation models, so as to better understand how our model outperforms others. Table 5 shows an example of codes produced from Django dataset. We find that the model based on depth-first traversal tends to generate the wrong function names, while TRANX′ prefers to makes a mistake on the generation of argument. Although using various strategies, most of models are unable to generate completely correct code. By contrast, our model ML-TRANX absorbs advantages of models with decoders based on different traversals and succeed in overcoming all these problems to generate more accurate code.

Related Work

With the rapid development of deep learning, neural network based models have now become dominant in code generation. In this aspect, Ling et al., (2016) considered code generation as a conditional text generation task, which can be solved by a neural sequence-to-sequence model. To exploit syntactic and semantic constraints of the programs, both of which are important for code generation, more researchers resorted to neural Seq2Tree models transforming each NL utterance into a sequence of AST based grammar actions. For example, Yin and Neubig (2017; 2018) proposed Seq2Tree models to generate tree-structured MRs using a series of tree-construction actions. Then, Sun et al. (2019; 2020) introduced CNN network and Transformer architecture to handle the long dependency problem. From a different perspective, Yin and Neubig (2019) explored reranking an N-best list of candidate results for adequacy and coherence of the programs. Meanwhile, other approaches have been extensively investigated. Wei et al., (2019a) applied dual training to jointly model code summarization and code generation, where the specific intuitive correlation between these two tasks can be fully leveraged to benefit each other.

Significantly different from the above work, we further explore the Seq2Tree model with breadth-first traversal based decoding. More importantly, we introduce mutual learning based model training framework to iteratively enhance multiple models via knowledge distillation. Mutual learning has been successfully used in image recognition (Zhang et al. 2018b; Furlanello et al. 2018; Lan, Zhu, and Gong 2018), machine translation (Zeng et al. 2019), machine reading comprehension (Liu et al. 2020), and sentiment analysis (Xue, Zhang, and Zha 2020). However, to the best of our knowledge, our work is the first attempt to explore this approach for code generation. Among all models for code generation, the most related to our is (Alvarez-Melis and Jaakkola 2017). This work proposed to use doubly-recurrent decoders (vertical and horizontal LSTMs) trained on ASTs, of which outputs can be concatenated to exploit context in different dimensions. Compared with this work, ours still significantly differs from it in following aspects: 1) Our approach only affects the model training while has no impact on the model testing; 2) During the model testing, David Alvarez-Melis and Jaakkola. (2017) simultaneously use two LSTMs with more parameters while we only use one LSTM decoder, and 3) Experimental results show both of our enhanced models are able to surpass (Alvarez-Melis and Jaakkola 2017).

Conclusion and Future Work

In this paper, we have explored the Seq2Tree model with breadth-first traversal based decoding and proposed a mutual learning based model training framework for code generation, where the models with different traversals based decodings can be enhanced simultaneously. Experimental results and in-depth analyses on several commonly-used datasets strongly demonstrate the effectiveness and generality of our proposed framework.

In the future, we plan to refine our framework via self-distillation (Wei et al. 2019b). Moreover, we will explore asynchronous bidirectional decoding (Zhang et al. 2018a; Su et al. 2019)to combine advantages of code generation models with different traversals based decodings (preorder traversal vs breadth-first traversal).

Acknowledgements

The project was supported by National Natural Science Foundation of China (No. 62036004, No. 61672440), Natural Science Foundation of Fujian Province of China (No.2020J06001) and College-level Undergraduate Innovation Training Project of School of Informatics Xiamen University (No.2020Y1071).

References

Alvarez-Melis, D.; and Jaakkola, T. S. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR 2017*.

Beltagy, I.; and Quirk, C. 2016. Improved Semantic Parsers For If-Then Statements. In *ACL 2016*, pages 726–736.

Clark, K.; Luong, M.; Khandelwal, U.; Manning, C. D.; and Le, Q. V. 2019. BAM! Born-Again Multi-Task Networks for Natural Language Understanding. In *ACL 2019*, pages 5931–5937.

Dong, L.; and Lapata, M. 2016. Language to Logical Form with Neural Attention. In *ACL 2016*, pages 33–43.

Dong, L.; and Lapata, M. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *ACL 2018*, pages 731–742.

Furlanello, T.; Lipton, Z. C.; Tschannen, M.; Itti, L.; and Anandkumar, A. 2018. Born-Again Neural Networks. In *ICML 2018*, pages 1602–1611.

Lan, X.; Zhu, X.; and Gong, S. 2018. Knowledge Distillation by On-the-Fly Native Ensemble. In *NeurIPS 2018*, pages 7528–7538.

Liang, P.; Jordan, M. I.; and Klein, D. 2011. Learning Dependency-Based Compositional Semantics. In *ACL 2011*, pages 590–599.

Ling, W.; Blunsom, P.; Grefenstette, E.; Hermann, K. M.; Kociský, T.; Wang, F.; and Senior, A. W. 2016. Latent Predictor Networks for Code Generation. In *ACL 2016*, pages 599–609.

Liu, X.; Liu, K.; Li, X.; Su, J.; Ge, Y.; Wang, B.; and Luo, J. 2020. An Iterative Multi-Source Mutual Knowledge Transfer Framework for Machine Reading Comprehension. In *IJCAI*, pages 3794–3800.

Quirk, C.; Mooney, R. J.; and Galley, M. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *ACL 2015*, pages 878–888.

Rabinovich, M.; Stern, M.; and Klein, D. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL 2017*, pages 1139–1149.

Romero, A.; Ballas, N.; Kahou, S. E.; Chassang, A.; Gatta, C.; and Bengio, Y. 2015. FitNets: Hints for Thin Deep Nets. In *ICLR 2015*.

Sennrich, R.; Birch, A.; Currey, A.; Germann, U.; Haddow, B.; Heafield, K.; Barone, A. V. M.; and Williams, P. 2017. The University of Edinburgh's Neural MT Systems for WMT17. In *WMT 2017*, pages 389–399.

Shin, E. C. R.; Allamanis, M.; Brockschmidt, M.; and Polozov, A. 2019. Program Synthesis and Semantic Parsing with Learned Code Idioms. In *NeurIPS 2019*, pages 10824–10834.

Su, J.; Zhang, X.; Lin, Q.; Qin, Y.; Yao, J.; and Liu, Y. 2019. Exploiting reverse target-side contexts for neural machine translation via asynchronous bidirectional decoding. *Artif. Intell.* 277.

Sun, Z.; Zhu, Q.; Mou, L.; Xiong, Y.; Li, G.; and Zhang, L. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *AAAI 2019*, pages 7055–7062.

Sun, Z.; Zhu, Q.; Xiong, Y.; Sun, Y.; Mou, L.; and Zhang, L. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *AAAI 2020*, pages 8984–8991.

Wei, B.; Li, G.; Xia, X.; Fu, Z.; and Jin, Z. 2019a. Code Generation as a Dual Task of Code Summarization. In *NeurIPS 2019*, pages 6559–6569.

Wei, H.-R.; Huang, S.; Wang, R.; Dai, X.-y.; and Chen, J. 2019b. Online Distilling from Checkpoints for Neural Machine Translation. In *ACL 2019*, pages 1932–1941.

Xu, F. F.; Jiang, Z.; Yin, P.; Vasilescu, B.; and Neubig, G. 2020. Incorporating External Knowledge through Pretraining for Natural Language to Code Generation. In *ACL 2020*, pages 6045–6052.

Xue, Q.; Zhang, W.; and Zha, H. 2020. Improving Domain-Adapted Sentiment Classification by Deep Adversarial Mutual Learning. In *AAAI 2020*, pages 9362–9369.

Yin, P.; and Neubig, G. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL 2017*, pages 440–450.

Yin, P.; and Neubig, G. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *EMNLP 2018*, pages 7–12.

Yin, P.; and Neubig, G. 2019. Reranking for Neural Semantic Parsing. In *ACL 2019*, pages 4553–4559.

Zeng, J.; Liu, Y.; Su, J.; Ge, Y.; Lu, Y.; Yin, Y.; and Luo, J. 2019. Iterative Dual Domain Adaptation for Neural Machine Translation. In *EMNLP-IJCNLP*, pages 845–855.

Zettlemoyer, L. S. 2009. Learning to map sentences to logical form. Ph.D. thesis., Massachusetts Institute of Technology, Cambridge, MA, USA .

Zettlemoyer, L. S.; and Collins, M. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *EMNLP-CoNLL 2007*, pages 678–687.

Zhang, X.; Su, J.; Qin, Y.; Liu, Y.; Ji, R.; and Wang, H. 2018a. Asynchronous Bidirectional Decoding for Neural Machine Translation. In McIlraith, S. A.; and Weinberger, K. Q., eds., *AAAI*, 5698–5705.

Zhang, Y.; Xiang, T.; Hospedales, T. M.; and Lu, H. 2018b. Deep Mutual Learning. In *CVPR 2018*, pages 4320–4328.