Antecedent Predictions Are More Important Than You Think: An Effective Method for Tree-Based Code Generation

Yihong Dong[1], Ge Li[1,*], Xue Jiang[1] and Zhi Jin[1]

[1]Key Lab of High Confidence Software Technology, MoE (Peking University)
dongyh@stu.pku.edu.cn, lige@pku.edu.cn, jiangxue@stu.pku.edu.cn, and zhijin@pku.edu.cn

Abstract. Code generation focuses on automatically converting natural language (NL) utterances into code snippets. Sequence-to-tree (Seq2Tree) approaches are proposed for code generation with the aim of ensuring grammatical correctness of the generated code. These approaches generate subsequent Abstract Syntax Tree (AST) nodes based on the preceding predictions of AST nodes. However, existing Seq2Tree approaches tend to treat both antecedent predictions and subsequent predictions equally, which poses a challenge for models to produce accurate subsequent predictions if the antecedent predictions are incorrect under the constraints of the AST. Given this challenge, it is necessary to pay more attention to antecedent predictions compared to subsequent predictions. To this end, this paper proposes a novel and effective method, named Antecedent Prioritized (AP) Loss, which prioritizes antecedent predictions by leveraging the position information of the generated AST nodes. We design an AST-to-Vector (AST2Vec) method that maps AST node positions to two-dimensional vectors, thereby modeling the position information of AST nodes. To evaluate the effectiveness of our proposed loss, we implement and train an Antecedent Prioritized Tree-based code generation model called APT. Experiments on four benchmark datasets demonstrate that with better antecedent predictions and accompanying subsequent predictions, APT achieves significant improvements, indicating the superiority and generality of our proposed method.

1 Introduction

Code generation is an essential generation task in the field of natural language processing (NLP) and software engineering [20, 24, 7, 8, 18, 35, 36, 16], which deals with automatically generating a piece of executable code from NL utterances. In recent years, a series of Seq2Tree models have made remarkable achievements for code generation [2, 38, 1, 39, 27, 29, 28, 33, 11, 14, 43, 21]. Specifically, given an NL utterance input, instead of outputting a sequence of code tokens directly, the Seq2Tree model outputs a sequence of AST actions. These AST actions correspond to AST nodes in a traversal method based on the abstract syntax description language (ASDL) grammar, and can eventually be converted into code via a certain deterministic function [1]. Depending on AST structure and ASDL grammar, Seq2Tree models ensure the grammatical correctness of gener-
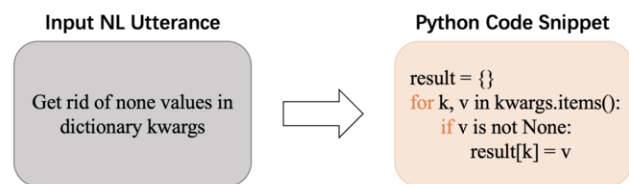
Figure 1: An illustration of code generation: Given a NL utterance as input, the task is to convert it into corresponding code snippet.

ated code. Furthermore, AST structure and ASDL grammar also help Seq2Tree models shrink the search space, i.e., antecedent predictions will constrain subsequent predictions, to generate well-formed codes.

Despite the success of the above models, they still neglect that antecedent predictions play an essential role in Tree-based code generation. The AST corresponding to code is a structure of progressive derivation from the root node to leaf nodes, and child nodes are derivations of parent nodes. Under the constraints of AST structure and ASDL grammar, parent nodes can easily determine the type, order, and even number of child nodes. When the output node sequence of Seq2Tree models is based on the pre-order traversal of AST[2], the parent node is always ahead of its child nodes. Consequently, when the antecedent parent node is wrongly generated, Seq2Tree models can barely predict subsequent child nodes correctly, which dramatically impacts the effectiveness of Seq2Tree models.

In this paper, we propose a novel loss function, namely AP Loss, that helps models prioritize antecedent predictions during training, so that generating more accurate antecedent predictions and accompanying subsequent predictions during inference. Intrinsically, AP Loss is a dynamically scaled cross entropy loss containing scaling and magnitude factors. The scaling factor is related to the position of the traversal order of AST nodes, and the magnitude factor controls the magnitude of the total loss function. Intuitively, AP Loss automatically reduces the contribution of susceptible subsequent predictions and focuses on more critical antecedent predictions during training. Therefore, we propose an AST2Vec method that models AST nodes with vectors to distinguish between antecedent nodes and subsequent nodes in AST. We implement and train an Antecedent Prioritized Tree-based code generation model called APT, utilizing

---

[1] According to the compiler theory, AST nodes can be transformed into code by extracting all leaf nodes and some key nodes of AST, which is the deterministic function.

[2] Note that breadth-first traversal has the same conclusion, and we use pre-order traversal as examples in this paper.
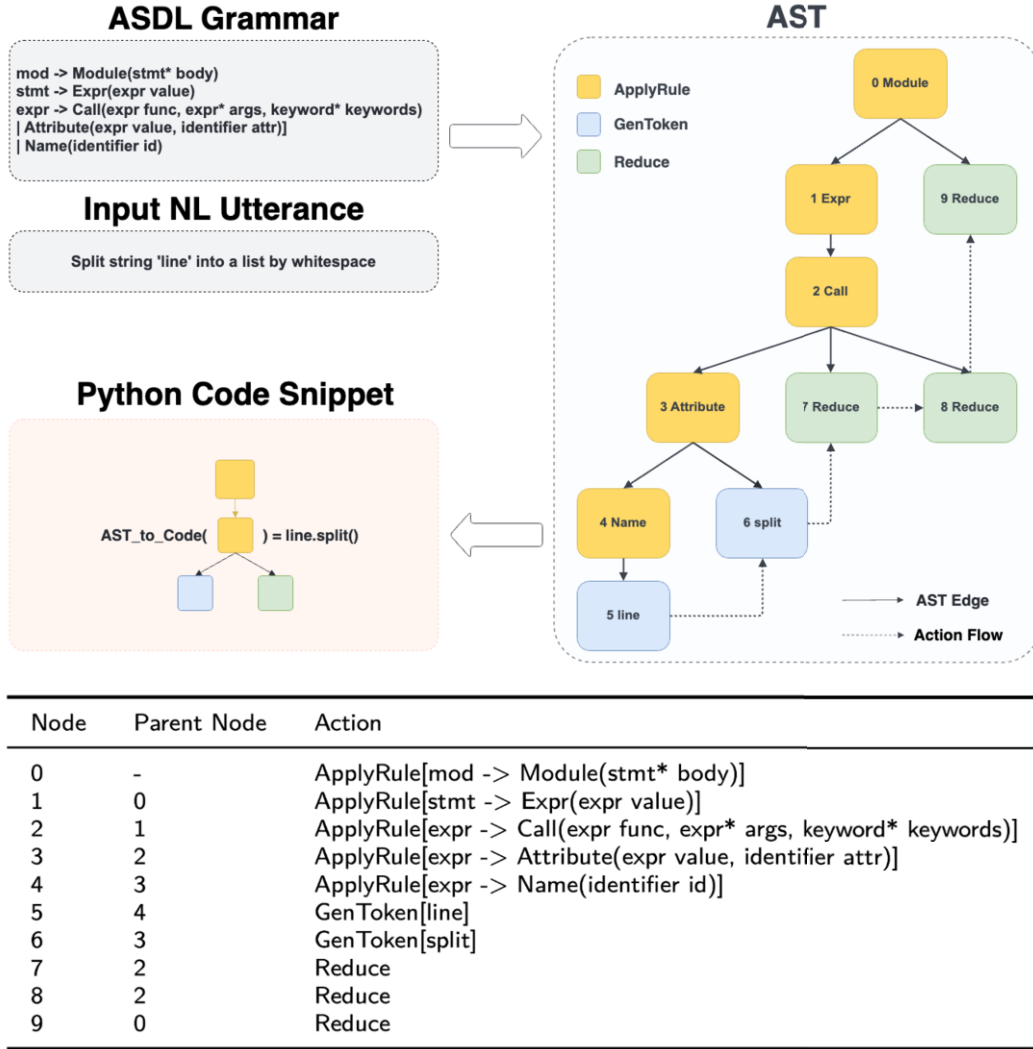
Figure 2: The procedure of code generation by Seq2Tree models. The order of nodes is based on the pre-order traversal of AST.

the AP Loss. To demonstrate the efficacy of our proposed method, we conduct extensive experiments on several benchmark datasets. Experimental results show that APT outperforms the state-of-the-art (SOTA) Seq2Tree methods, including ML-TRANX [33], TRANX-RL [12], and ASED [11], on four benchmark datasets. Furthermore, we note that AP Loss is suitable for not only different traversal methods but also different forms of weights, and maintains an advantage over Seq2Tree models, which indicates its generality.

The main contribution of this paper can be summarized as follows:

- We propose that antecedent predictions are more critical than subsequent predictions in the tree-based code generation framework.
- We propose AP Loss, an effective method for Seq2Tree models, which prioritizes antecedent predictions by leveraging the position information of generated AST nodes. Additionally, we introduce the AST2Vec method, which represents the position information of AST nodes through two-dimensional vectors and has the capability to reconstruct the corresponding AST.
- We instantiate an Antecedent Prioritized Tree-based code generation model, referred to as APT. APT outperforms the SOTA Seq2Tree methods on four benchmark datasets. Extensive experimental results demonstrate the effectiveness and generality of our proposed method.

2   Motivation

Source code is specialized text with strict syntactic rules and specific structural patterns. Certain key statements significantly impact the structure of code. Some famous examples such as *for*, *while*, and *if*, are shown in an example of code generation in Fig. 1. When the code is converted to AST with ASDL grammar, these key statements are presented as dominant antecedent nodes [3] in the AST. An AST and its corresponding ASDL grammar are depicted in Fig. 2. In this illustration, the yellow nodes represent antecedent nodes that determine the tree's structure. From this observation, we further summarize several properties that AST nodes exhibit under the constraints of ASDL grammar:

1. The parent node can determine the left side of the production rule of the child node, e.g., since the right side of the production rule of node 0 is *Module(stmt* body)*, the left side of the production rule of node 1 should be *stmt*.
2. The parent node is able to control the number and order of the child node. For instance, the right side of the production rule of

---

[3] An antecedent node of a node is either the parent of this node or the parent of some ancestors of this node.
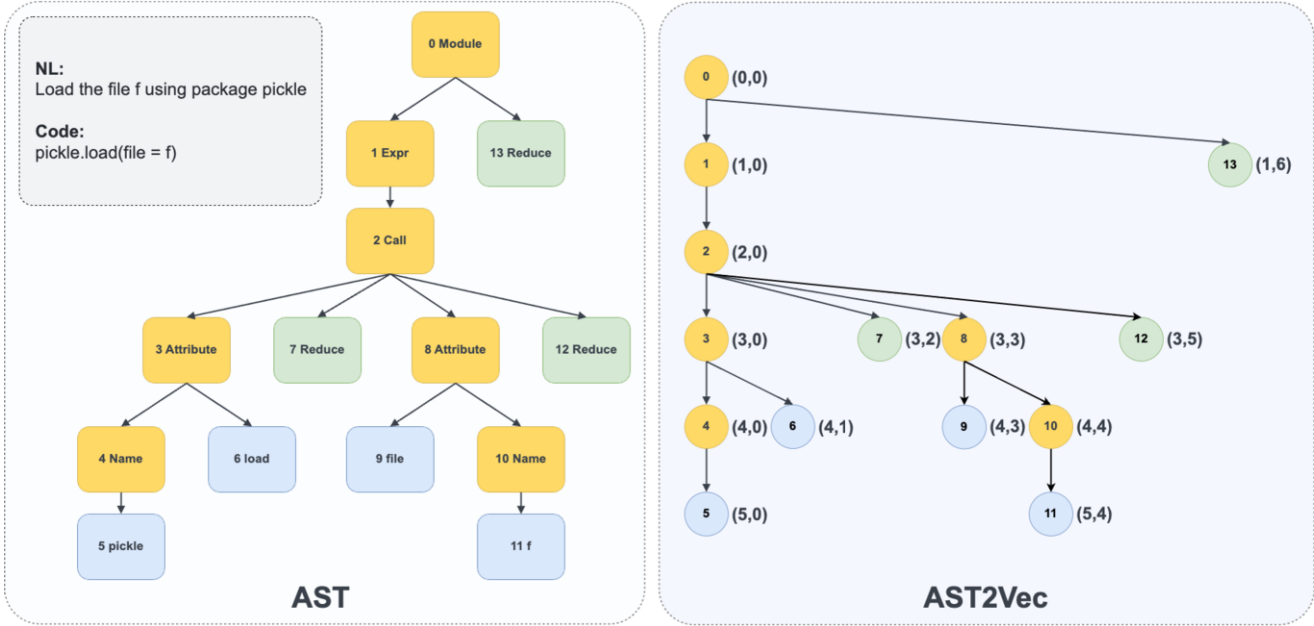
Figure 3: An Example of AST2Vec.

node 3 is *Attribute(expr value, identifier attr)*, therefore node 3 has two child nodes, the first is *expr* and the second is *identifier*.

3. The parent node allows for a decision of the type of the child node. For example, because the right side of the production rule of node 4 is *Name(identifier id)*, the type of node 5 has to be the id of the identifier. Similarly, the type of node 6 has to be the attribute of the identifier.

The properties of an AST underscore the critical role played by the antecedent nodes in code generation. Should the antecedent node's prediction prove inaccurate, subsequent node predictions will be adversely impacted, resulting in incorrect predictions. Fig. 2 shows the procedure of converting an input NL utterance into the corresponding python code snippet by Seq2Tree models by generating actions to construct an intermediate representation AST. During training, the Seq2Tree model generates the subsequent action predicated on the previous ground-truth action at each training step. Conversely, during testing, the model lacks knowledge of the ground truth action and proceeds to generate the next action based on the previously generated action. This discrepancy results in a non-negligible inconsistency between the training and inference phases. It is incredibly challenging to generate an accurate subsequent prediction based on incorrect antecedent predictions under AST structure and ASDL grammar constraints. Thus, the cost of producing wrong antecedent predictions is quite expensive during inference. To this end, in this paper, we propose an effective method that pays more attention to pre-generation during training and alleviates the negative effect of training inconsistency for Tree-based code generation methods.

3   APT

In this section, we first propose a method of representing AST node positions as vectors, called AST2Vec. Then, we describe the principle and architecture of our base model. Finally, we introduce AP Loss in detail.

3.1   AST2Vec

The sequence with pre-order traversal cannot represent the position information of AST nodes in AST well. Therefore, we consider adopting two-dimensional vectors to represent the position information of AST nodes. These vectors need to have the ability to reconstruct AST, which has to comply with the following rules:

- The vector corresponding to each AST node should be unique;
- The capability to determine a unique parent node for any child node based on vectors.

Therefore, we design an AST2Vec method that conforms to these two rules. Specifically, we perform a pre-order traversal for an AST starting from the root node, and the vertical coordinate of the child node is that of its parent node + 1. If the parent node of the current node is traversed for the first time, and its horizontal coordinate is the horizontal coordinate of its parent node. If the parent node of the current node is traversed for the second and more times, its horizontal coordinate is the maximum of horizontal coordinates of all traversed nodes + 1. We provide the pseudocode for our AST2Vec method in Algorithm 1 and an example of AST2Vec is shown in Fig. 3.

Representing ASTs with vectors based on AST2Vec offers two significant advantages:

1. *We can reconstruct the AST according to the corresponding vectors.* According to the algorithm used to generate the following-sibling nodes, each node is assigned a unique vector that does not overlap with any other nodes. This is achieved by setting the horizontal coordinate of a node to the maximum horizontal coordinate of all preceding nodes plus one, while ensuring that child nodes of following-sibling nodes have greater vertical coordinates. Furthermore, to determine the parent node of a given node, we look for the node whose vertical coordinate is one less than that of the current node, and whose horizontal coordinate is the first one less than or equal to the current node's horizontal coordinate. For example, in Fig. 3, the parent node of nodes 9 and 10 must be node 8, rather than node 7 or node 3.

Algorithm 1 AST2Vec

Require: The AST $\mathcal{T}$ or the pre-order traversal action sequence of AST **a**.
Ensure: The vectors **v**.
  1: Initial the vectors **v**, the record set $R = \emptyset$, and the maximum $maxn = 0$.
  2: **if a** is not exist **then**
  3:     **a** $\leftarrow$ the pre-order traversal of $\mathcal{T}$.
  4: **end if**
  5: **for** a_t in **a do**
  6:     **if** a_t is root node **then**
  7:         $v_t = (0, 0)$.
  8:         continue.
  9:     **end if**
10:     $a_p \leftarrow$ the parent node of a_t.
11:     **if** $a_p \notin R$ **then**
12:         $v_t = (v_p.d_1 + 1, v_p.d_2)$.
13:         add a_p to R.
14:     **else**
15:         $maxn = maxn + 1$.
16:         $v_t = (v_p.d_1 + 1, maxn)$.
17:     **end if**
18: **end for**
19: **return v**
---

2. *The subtraction of vectors satisfies the Triangle Law* [4], *which can reflect the relationship between AST nodes.* Concretely, the vector connecting two nodes can be represented by calculating the difference between their respective vector representations. For instance, in Fig. 3, the vector representation for node 8 is $(3, 3)$, while that for node 9 is $(4, 3)$. Therefore, the vector from node 8 to node 9 is computed as follows: $(4, 3) - (3, 3) = (1, 0)$.

In conclusion, the unique vector representation assigned to each node ensures that the AST can be accurately reconstructed from the corresponding vectors, while also allowing for easy determination of parent-child relationships between nodes. Further, the use of vector subtraction to calculate the difference between vectors satisfies the Triangle Law, allowing for an accurate reflection of the relationships between AST nodes. These benefits demonstrate the effectiveness of the AST2Vec approach in capturing the structural information of ASTs in a concise and useful way.

3.2 Base Model

Our proposed AP Loss is suitable for all Seq2Tree model. Here we adopt TRANX [39] as our base model, given its widespread usage and demonstrated competitive performance in code generation. Numerous Seq2Tree methods are based on and improve upon TRANX, including ML-TRANX [33], TRANX-RL [12], ASED [11], etc.

In Fig. 2, TRANX first outputs a sequence of actions to construct AST in pre-order traversal based on ASDL grammar and the input NL utterance. Then a given function AST_to_Code(*) is called to map the generated intermediate AST into code. Concretely, three types of ASDL grammar-based actions are available:

APPLYRULE[c] actions apply a constructor c to the composite field of a node with the same type as c, generating the child node using the fields in c.

REDUCE actions complete the generation of the child node for a field that has optional(?) or multiple($\star$) cardinalities.

GENTOKEN[v] actions expand the field of a node to generate a token v.

TRANX uses an attentional encoder-decoder framework with augmented recurrent connections to interpret the topology of AST. The encoder of TRANX is a Bidirectional Long Short-term Memory (BiLSTM) network, and the decoder of TRANX is an LSTM network. Given an input NL utterance **x** of n tokens, i.e., $\{x_i\}_{i=1}^n$, the BiLSTM encoder learns vectorial representations $\{\mathbf{z}\}_{i=1}^n$. Then, at each step t, the LSTM decoder generates hidden state $\mathbf{s}_t$ as:

$$\mathbf{s}_t = f_{\text{LSTM}}([\mathbf{e}_{t-1} : \tilde{\mathbf{s}}_{t-1} : \mathbf{p}_t], \mathbf{s}_{t-1}), \qquad (1)$$

where f_{LSTM} is the LSTM decoder transition function, $[:]$ denotes the concatenation of vectors, $\mathbf{e}_{t-1}$ and $\mathbf{p}_t$ denote the embedding of previous action and parent action, respectively. We define the attentional vector $\tilde{\mathbf{s}}_t$ as:

$$\tilde{\mathbf{s}}_t = \tanh(\mathbf{W}_c[\mathbf{c}_t : \mathbf{s}_t]), \qquad (2)$$

where $\tanh$ is the hyperbolic tangent function, $\mathbf{W}_c$ is a parameter matrix, $\mathbf{c}_t$ denotes the context vector weighted by encoding representations $\{\mathbf{z}\}_{i=1}^n$ using attention.

Finally, the probability of an APPLYRULE[c] action with embedding $\mathbf{e}_c$ at each step t is:

$$p(a_t = \text{APPLYRULE}[c] \mid a_{<t}, \mathbf{x}) = \text{softmax}(\mathbf{e}_c^T \mathbf{W} \tilde{\mathbf{s}}_t), \quad (3)$$

where softmax is the softmax function, $\mathbf{e}_c^T$ denotes the transpose of embedding $\mathbf{e}_c$, and $\mathbf{W}$ is another parameter matrix. Note that REDUCE is considered as a special APPLYRULE action. For the GENTOKEN action, its probability is defined as:

$$
\begin{aligned}
&p(a_t = \text{GENTOKEN}[v] \mid a_{<t}, \mathbf{x}) \\
&= p(\text{gen} \mid a_t, \mathbf{x}) p(v \mid \text{gen}, a_t, \mathbf{x}) \\
&+ p(\text{copy} \mid a_t, \mathbf{x}) p(v \mid \text{copy}, a_t, \mathbf{x}),
\end{aligned}
$$

where

$$
\begin{aligned}
p(\text{gen} \mid a_t, \mathbf{x}) &= \text{softmax}(\mathbf{W} \tilde{\mathbf{s}}_t), \\
p(\text{copy} \mid a_t, \mathbf{x}) &= 1 - p(\text{gen} \mid a_t, \mathbf{x}), \\
p(v \mid \text{gen}, a_t, \mathbf{x}) &= \text{softmax}(\mathbf{e}_v^T \mathbf{W} \tilde{\mathbf{s}}_t), \\
p(x_i \mid \text{copy}, a_t, \mathbf{x}) &= \text{softmax}(\mathbf{z}_i^T \mathbf{W} \tilde{\mathbf{s}}_t).
\end{aligned}
$$

Similar to other Seq2Tree models, given a corpus $(\mathbf{x}, \mathbf{a})$, TRANX is trained to minimize the cross entropy loss:

$$\mathcal{L}_{\text{ce}}(\mathbf{x}, \mathbf{a}; \theta) = -\sum_{t=1}^T \log p(a_t \mid a_{<t}, \mathbf{x}; \theta), \qquad (4)$$

where T is the total number of actions and θ denotes parameters of the model.

3.3 Antecedent Prioritized Loss

We extend the standard cross entropy loss (4) to AP Loss by exploiting the position information of AST traversal order. Specifically, a scaling factor $f(\mathbf{a}, t)^{-\gamma}$ is added with respect to AST action sequence and the action index t, and the AP Loss is defined as:

$$\mathcal{L}_{\text{ap}}(\mathbf{x}, \mathbf{a}; \theta) = -\sum_{t=1}^T f(\mathbf{a}, t)^{-\gamma} \log p(a_t \mid a_{<t}, \mathbf{x}; \theta), \qquad (5)$$

$$f(\mathbf{a}, t) = \text{Norm}(\text{AST2Vec}(\mathbf{a})_t) \qquad (6)$$

---
[4] To find the difference of two vectors $\boldsymbol{b}_1 - \boldsymbol{b}_2$ that are coinitial, just draw a vector from the tail of $\boldsymbol{b}_2$ to the tail of $\boldsymbol{b}_1$.
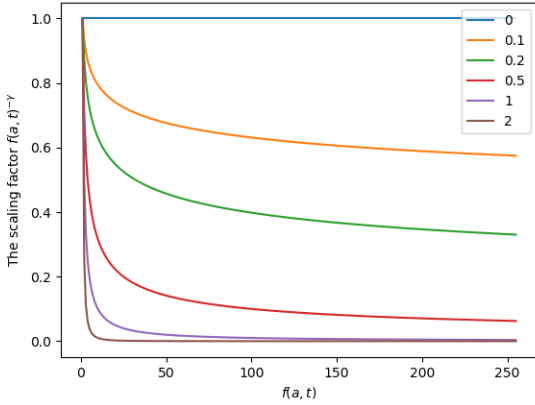
Figure 4: The effect of varying γ on the scaling factor $f(\mathbf{a}, t)^{-\gamma}$.

where the tunable hyperparameter $\gamma \geq 0$, the function AST2Vec uses Algorithm 1, and $\text{Norm}(\text{AST2Vec}(\mathbf{a})_t)$ indicates the norm of the vector $\text{AST2Vec}(\mathbf{a})_t$. The scaling factor $f(\mathbf{a}, t)^{-\gamma}$ decreases as $f(\mathbf{a}, t)$ increases, which helps models down-weight actions with large $f(\mathbf{a}, t)$ and thus focus training on actions with small $f(\mathbf{a}, t)$, i.e., antecedent actions. The rationale for this is that, when the AST is mapped to the corresponding vectors using AST2Vec, the vector norm of the parent node is always smaller than that of the child node, as shown in Fig. 3. This property enables us to prioritize training on antecedent actions by reducing the contribution of descendant actions through the scaling factor. In summary, the hyperparameter γ provides control over the weighting of different actions in our model, and the scaling factor helps us prioritize antecedent actions by reducing the contribution of descendant actions during training.

Fig. 4 displays the scaling factor for several values of $\gamma \in [0, 2]$. When $\gamma = 0$, the AP Loss ($\mathcal{L}_{\text{ap}}$) reduces to the standard cross entropy loss ($\mathcal{L}_{\text{ce}}$). It is advisable to avoid selecting $\gamma > 1$, as the resulting scaling factor $f(\mathbf{a}, t)^{-\gamma}$ can vary dramatically. For instance, when $\gamma = 2$, an action with index 10 would have a loss that is $100\times$ lower than that of $\mathcal{L}_{\text{ce}}$. This could lead models to disregard subsequent predictions completely, which is undesirable. To achieve a balance between the loss of antecedent and subsequent predictions, a suitable scaling factor should be chosen. Thus, we recommend selecting γ from the range 0.1 to 0.5. It is worth noting that implementing the AP Loss in code is relatively straightforward and we provide the pseudocode of AP Loss in Fig. 5.

In practice, we insert an α-adjusted magnitude factor in AP Loss:

$$\mathcal{L}_{\text{ap}}(\mathbf{x}, \mathbf{a}; \theta) = -\alpha \sum_{t=1}^{T} t^{-\gamma} \log p\left(a_t \mid a_{<t}, \mathbf{x}; \theta\right). \quad (7)$$

We adopt the above form in our experiments because it yields improvements over the non-α-adjusted form (5). Generally, α can be chosen from positive constants.

```
def AP_Loss(actions_prob, actions, α, γ):
    weight = [αf(actions, t)^−γ for t in range(1, actions_prob.size()+1)]
    return − weight * actions_prob.log()
```

Figure 5: Pseudocode for AP Loss.

4 Experiment Setup

4.1 Datasets

We conduct experiments on four public benchmark datasets as follows: **GEO** [41] consists of 880 US geographical NL utterances and their corresponding code defined in lambda logical forms. **ATIS** [10] is a collection of 5,410 paired flight data, which contains the NL description of flight information and their corresponding lambda calculus code like GEO. **DJANGO** [23] is a popular code generation dataset consisting of NL-annotated python2 code extracted from Django Web framework. **CONALA** [37] contains 2,879 real-world data of manually annotated NL questions and their Python3 solutions on STACK OVERFLOW, which is more difficult because of its complex and extensive code composition.

The detailed statistics of the above datasets are shown in Table 1.

Table 1: Statistics of benchmark datasets.

Dataset	Examples Num			Avg Length		
	Train	Dev	Test	NL	Code	AST
GEO	600 (480)	- (120)	280	7.4	19.1	19.3
ATIS	4,473	491	448	10.5	28.1	31.5
DJANGO	16000	1000	1805	14.1	10.6	14.4
CONALA	2,175	200	500	10.2	15.1	23.2

4.2 Baselines

To exhibit the effectiveness of our proposed method, we primarily compare APT with several competitive Seq2Tree methods, which can guarantee the syntactic correctness of generated code, as follows:

SEQ2TREE [2] uses a general attention-enhanced encoder-decoder model to generate logical forms by conditioning output trees.

ASN [25] outputs AST constructed by a decoder whose modular structure is dynamically determined, parallel to the structure of output trees.

COARSE2FINE [3] first generates a rough meaning sketch abstracted from low-level information, and then fills in missing detail.

TREEGEN [29] uses Transformer instead of LSTM to alleviate the long-dependency problem and adopts an AST reader to incorporate both grammar rules and AST structures.

TRANX [39] is our base model which has detailed descriptions in Section 3.2. TRANX and TRANX' are the versions of the pre-order and breadth-first traversals of TRANX, respectively.

ML-TRANX [33] adopts a mutual learning framework to train models for different traversals based decodings jointly.

TRANX-RL [12] makes use of a context-based branch selector to dynamically determine optimal branch expansion orders for multi-branch nodes.

ASED [11] generates the current prediction with both the history and future information using an AST structure enhanced decoder.

Following the previous work [29, 33, 12], we mainly compare our proposed method to the SOTA Seq2Tree methods with similar parameter sizes to ensure the fairness of the comparison.

4.3 Metrics

To evaluate the effectiveness of different methods, we use two widely-used evaluation metrics: the exact matching accuracy (EM)

Table 2: The performance of APT compared with various baselines.

Model	GEO	ATIS	DJANGO	CONALA	
	EM	EM	EM	BLEU	EM
SEQ2TREE	86.1	84.6	-	-	-
ASN	87.1	85.9	-	-	-
COARSE2FINE	88.2	87.7	74.1	-	-
TREEGEN	89.6	89.1	-	-	-
TRANX	88.8 ± 1.0	87.6 ± 0.1	77.3 ± 0.4	24.35 ± 0.4	2.5 ± 0.7
ML-TRANX	89.2 ± 0.6	89.3 ± 0.3	$\mathbf{79.6 \pm 0.3}$	24.42 ± 0.8	2.2 ± 0.4
TRANX-RL	89.5 ± 1.2	89.1 ± 0.5	77.9 ± 0.5	25.47 ± 0.7	2.6 ± 0.4
ASED	89.8 ± 1.1	88.9 ± 0.7	79.2 ± 0.5	25.56 ± 0.6	2.8 ± 0.7
APT (ours)	$\mathbf{90.4 \pm 0.8}$	$\mathbf{89.8 \pm 0.7}$	$\mathbf{79.6 \pm 0.5}$	$\mathbf{28.02 \pm 0.7}$	$\mathbf{2.9 \pm 0.7}$

for all datasets and the corpus-level BLEU-4 for CONALA as a complement. The reason is that CONALA is a more challenging dataset compared to other datasets, EM may be too strict and can hardly reflect the performance of methods on it. As the AP Loss does not impact the inference time, the execution time of each method is not reported. It should be noted that none of the four publicly available benchmark datasets include test cases, thus the evaluation metric of "pass@k" is not applied.

4.4 Settings

We train our model for a maximum epoch of 80 with Adam [15] optimizer on a single GPU of Tesla V100-SXM2-32G. To ensure fair comparisons, we adopt the same experimental setup as TRANX [39]. To be specific, we set the sizes of word embedding, action embedding, and hidden states as 128, 128, and 256, respectively. The beam size is set to 5 for GEO and ATIS, while 15 for DJANGO and CONALA. According to the analysis in Sections 3.3 and 6.1, we pick $\gamma \in [0.1, 0.5]$ using validation set and set $\alpha = 2$ for all datasets. To mitigate the instability of the model training, we exhibit the average performance of the model running five times.

5 Experimental Results

In this section, comprehensive experiments are carried out to answer the following research questions:

RQ1: How does the proposed APT perform compared to various competitive and SOTA Seq2Tree methods without pre-training?

RQ2: What is the performance of different traversal methods of the base model based on AP Loss?

RQ3: What is the performance of the base model based on AP Loss with different forms?

RQ4: What is the performance of different Seq2Tree methods based on AP Loss?

RQ5: How does schedule sampling compare to AP loss in dealing with problems that help models focus on antecedent prediction?

RQ6: How does the critical hyperparameter γ in (5) affect APT?

RQ7: How does APT generate codes with the help of AP Loss?

RQ1, RQ5, and RQ7 exhibit the effectiveness of APT and AP Loss, while RQ2, RQ3, RQ4, and RQ6 display the generality of them.

5.1 APT vs. Various Baselines

Table 2 lists the comparison of APT and various baselines on four benchmark datasets. With the help of AP Loss, our APT achieves significant improvements compared with base modal TRANX [39]

across all datasets. In particular, APT relatively improves 15.1% BLEU on CONALA dataset compared to TRANX, which demonstrates the gain effect of AP loss. APT surpasses all TRANX variants and other Seq2Tree models, demonstrating notable performance. It is important to highlight that EM is a stringent evaluation metric, and the improvements achieved by APT are more pronounce than previous works. As a result, these experimental results indicate that letting the Seq2Tree model concentrate on the predictions of the antecedent AST nodes during training helps code generation in inference.
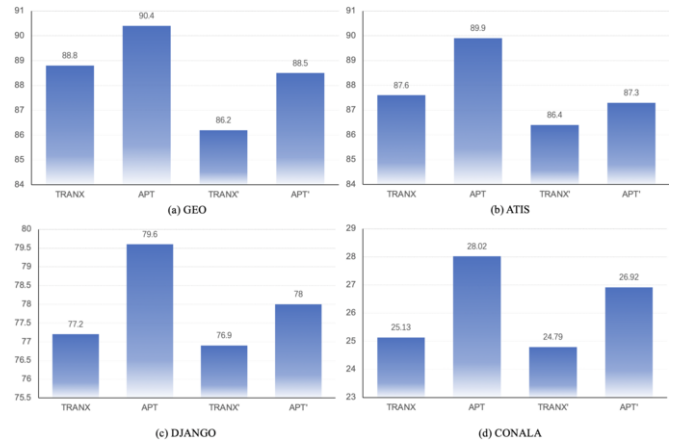

Figure 6: The performance of AP Loss based on the pre-order and breadth-first traversals of TRANX, i.e., APT and APT'. (BLEU for CONALA and EM for other datasets)

5.2 Pre-order vs. Breadth-first Traversal

As shown in Fig. 6, the performance of APT' (breadth-first traversals of TRANX based on AP Loss) is inferior to that of APT on all datasets, due to the difference in the respective traversal method of the base model. A possible reason is that the breadth-first traversal has some drawbacks. For example, back to Fig. 2, the attribute of the variable is output before the variable in breadth-first traversal, which increases the difficulty of prediction. Moreover, in Fig. 2, the breadth-first traversal, compared to the pre-order traversal, prioritizes the output of the Reduce actions and delays the production of GenToken actions, which is actually evaluated by metrics such as BLEU and EM. The code token belongs to the leaf node of AST and is generated last in the breadth-first traversal, which dramatically affects the effect. Therefore, we should trade off both antecedent and leaf nodes in the generation. Nevertheless, both the pre-order and breadth-first traversals of TRANX are significantly improved using

AP Loss. It implies that AP Loss can be applied to other reasonable traversal orders.
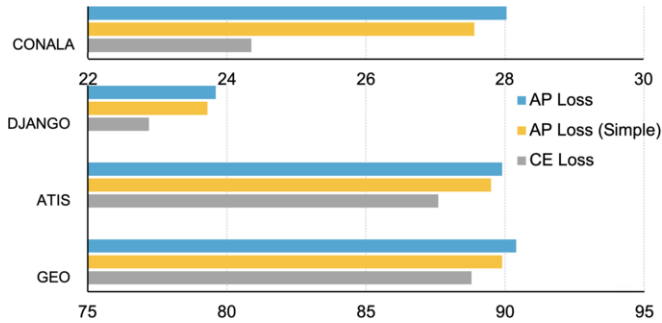

Figure 7: The performance of the base model based on AP Loss with different forms, where AP Loss (Simple) indicates $f(\mathbf{a}, t) = t$ in Eq. (5). (BLEU for CONALA and EM for other datasets)

5.3 AP Loss vs. AP Loss (Simple)

Note that $f(\mathbf{a}, t)$ in Eq. (5) can be any reasonable function reflecting the effect of the antecedent prediction. For example, $f(\mathbf{a}, t) = t$ [5]. We show the performance of the base model based on AP Loss with different forms for training in Fig. 7. Remarkably, utilizing the simplified function $f(\mathbf{a}, t) = t$, which represents the AP Loss (Simple), still achieves a remarkable improvement compared to $\mathcal{L}_{ce}$ on all datasets. This illustrates the importance of helping the model attach importance to antecedent predictions. We also find that the effect of using AP loss is significantly better than using AP Loss (Simple). This is because the position of the node in the pre-order traversal of AST only partially reflects its importance. The weight of sibling nodes in AP Loss (Simple) can vary widely, while that of sibling nodes in AP Loss is much smaller.

Table 3: Comparisons of Seq2Tree models and their use of AP Loss on CONALA.

Model	BLEU	EM
TRANX	24.35 ± 0.4	2.5 ± 0.7
+ AP Loss	$28.02 (\uparrow 15.1\%) \pm 0.7$	2.9 ± 0.7
ML-TRANX [33]	24.42 ± 0.8	2.2 ± 0.4
+ AP Loss	27.96 ± 0.7	2.6 ± 0.5
TRANX-RL [12]	25.47 ± 0.7	2.6 ± 0.4
+ AP Loss	27.88 ± 0.6	2.7 ± 0.5
ASED [11]	25.56 ± 0.6	2.8 ± 0.7
+ AP Loss	27.91 ± 0.7	2.9 ± 0.9

5.4 AP Loss with different Seq2Tree methods

We pick CONALA for the follow-up comparison of AP Loss with two reasons: 1) Python used by CONALA has a richer syntactic structure than Lambda code used by GEO and ATIS. 2) The average AST length of CONALA exceeds that of DJANGO by 60%.

In Table 3, we apply AP Loss to Seq2Tree models and compare their performance on CONALA. The experimental results demonstrate that all these Seq2Tree models achieve significant improvements using AP Loss. Moreover, the effect of AP loss on four Seq2Tree models is quite similar, probably because 1) we uniformly

---

[5] In pre-order traversal of AST node, the parent node precedes the child node.

adopt the hyperparameter γ used for TRANX without adjusting; 2) improved model performance is usually accompanied by better antecedent predictions.

Table 4: EM of the different percent of AST node sequences and code token sequences on CONALA.

Model	5%		10%		20%		50%	
	AST	code	AST	code	AST	code	AST	code
TRANX	45.2	17.4	26.4	8.6	11.8	4.2	5.8	2.6
+ AP Loss	49.0	23.0	30.6	14.6	14.8	7.0	9.2	4.8
ML-TRANX	46.8	18.2	27.6	9.6	12.4	3.8	5.6	2.4
+ AP Loss	49.8	22.8	30.8	14.0	14.6	6.6	8.8	4.4
TRANX-RL	47.0	20.0	28.2	11.0	13.4	4.6	5.4	3.0
+ AP Loss	48.8	22.6	30.2	13.6	14.8	6.6	8.8	4.6
ASED	47.8	17.2	28.6	9.2	13.2	4.8	5.8	3.2
+ AP Loss	49.8	22.8	29.8	13.8	15.4	6.8	8.6	4.8

In Table 4, we test the EM of antecedent AST node and code token sequence predictions on CONALA. The experimental results demonstrate that Seq2Tree methods achieve higher EM on the first 5%, 10%, 20%, and 50% of AST node sequences and code token sequences using AP Loss. It indicates that the Seq2Tree method can obtain better antecedent predictions with the help of AP Loss.

Table 5: Comparison of schedule sampling and AP Loss on CONALA, where p denotes the probability of using the label as input (TRANX is equivalent to the case of p=1), ED denotes exponential decay, e denotes the base of the exponent, and LD denotes linear decay.

| Model		Test Set		Training Set	
		BLEU	EM	BLEU	EM
TRANX		24.35	2.5	63.09	36.11
+ schedule sampling	p=0	22.94	1.6	61.44	37.90
	ED (e=0.9)	24.45	1.6	65.22	36.41
	ED (e=0.99)	25.59	2.2	**76.59**	**53.63**
	LD	25.14	2.0	66.85	37.95
+ AP Loss		**28.02**	**2.9**	68.52	44.35

5.5 AP Loss vs. Schedule Sampling

Schedule sampling is a conventional method employed to address the issue of inconsistency between the training and inference phases. In Table 5, we compare AP Loss with schedule sampling, utilizing TRANX as the base model. It becomes evident that schedule sampling indeed results in improved performance on the training set during the inference phase, in terms of both BLEU and EM scores. However, schedule sampling does not significantly enhance the model's focus on antecedent predictions, leading to a relatively minor impact on the test set.

5.6 Effects of γ

The coefficient γ serves as a crucial hyperparameter for regulating AP Loss. To examine the effect of γ on AP Loss, we present an analysis in Fig. 8. Optimal values for γ are found to be 0.4 for GEO and DJANGO datasets, and 0.1 for ATIS and CONALA datasets. Interestingly, we observe an inverse correlation between the best γ values and the average AST length, as presented in Table 1. This can be attributed to the fact that during the training phase, AP Loss must

Table 6: Case study examples on four benchmark datasets. The incorrect codes are marked in blue, while the first token of the counterparts is marked in red.

Dataset	Model	NL	Code
GEO	TRANX	What length is the r0?	r0 ✗
	APT		(len:i r0) ✔
	TRANX	What is the highest elev of c0?	(argmax $0 (and (place:t $0) (loc:t $0 c0)) c0) ✗
	APT		(argmax $0 (and (place:t $0) (loc:t $0 c0)) (elevation:i $0)) ✔
ATIS	TRANX	Is there ground transport from the ap0 into ci0 town?	(lambda $0 e (exists $1 (and (= (arrival_time $1) $0) (approx_arrival_time $1 ti0) (ground_transport $1) (to_city $1 ci0)))) ✗
	APT		(lambda $0 e (and (from_airport $0 ap0) (ground_transport $0) (to_city $0 ci0))) ✔
	TRANX	Tell me about the type of aircraft call an ac0.	(lambda $0 e (exists $1 (and (flight $1) (= (aircraft_code $1) $0) (aircraft_code $1 ac0)))) ✗
	APT		ac0 ✔
DJANGO	TRANX	Is length of list_ equals integer 0?	list_ = len(list_) + 0 ✗
	APT		if len(list_) == 0: pass ✔
	TRANX	Close zfile stream.	close(zfile.stream()) ✗
	APT		zfile.close() ✔
CONALA	TRANX	Get attribute str_0 from object var_0.	print(var_0.rstrip(str_0)) ✗
	APT		getattr(var_0, str_0) ✔
	TRANX	Running bash script str_0.	exec() ✗
	APT		subprocess.call(str_0, shell=True) ✔
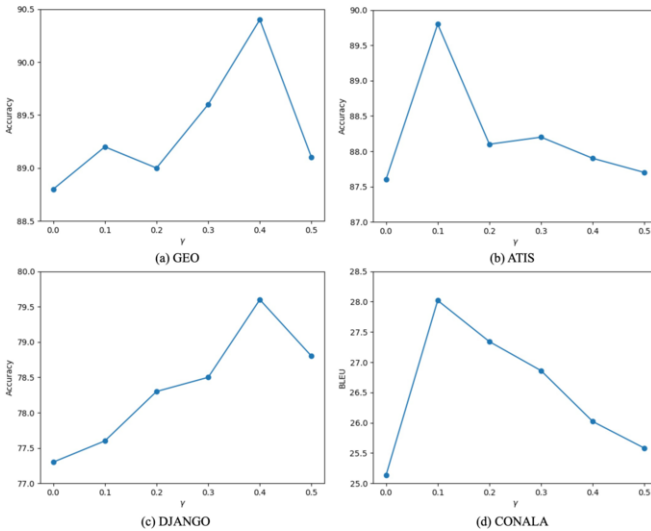
Figure 8: Performance of varying γ in (5) on four benchmark datasets.

appropriately increase the scaling factor $f(\mathbf{a}, t)^{-\gamma}$ by reducing γ for longer ASTs to prevent the neglect of later AST nodes.

5.7 Case Study

In Table 6, we present a comparison of the top-1 generated code from TRANX and APT on various datasets. It is evident that the majority of errors tend to occur at the beginning or in the leading part of the code, subsequently leading to further faults. As illustrated in Table 6, TRANX is prone to producing incorrect antecedent predictions, which poses a challenge for generating subsequent predictions while adhering to the constraints of AST. In contrast, our proposed model, APT, effectively mitigates the aforementioned issue by employing the AP Loss, thereby generating more accurate code.

6   Discussion

In this section, we mainly discuss the α selection and the computational overhead of AP Loss.

6.1 The Selection of α

For continuous scaling factor f, we can keep the magnitude of $\mathcal{L}_{ap}$ equal to the magnitude of original $\mathcal{L}_{ce}$ using the integral, which relates α to γ. For example, when $f(\mathbf{a}, t) = t$:

$$\begin{cases} \alpha = \dfrac{T}{\ln(T+1)}, & if \ \gamma = 1, \\ \alpha = (1-\gamma)T^{\gamma}, & otherwise. \end{cases} \quad (8)$$

With the help of (8), we only need to tune γ, and the appropriate α will be calculated automatically.

In Table 7, we vary α for AP Loss (Simple) with optimal γ. As α increases, the general trend of the performance of APT (Simple), i.e., the base model + AP Loss (Simple), rises and then falls on all datasets. When α is set to 'auto', APT (Simple) achieves the best performance in terms of EM for ATIS, DJANGO, and CONALA, and impressive results regarding EM for GEO and BLEU for CONALA. When we substitute the optimal γ and the corresponding average

Table 7: Varying α for AP Loss (Simple) with optimal γ, where 'auto' denotes α is calculated by (8).

α	GEO	ATIS	DJANGO	CONALA	
	EM	EM	EM	BLEU	EM
1	89.6	87.9	78.8	26.75	2.4
2	**89.9**	88.6	79.1	**27.56**	2.9
4	88.8	87.7	78.5	25.64	2.6
8	88.9	87.3	78.3	24.93	2.3
'auto'	89.7	**89.5**	**79.3**	27.12	**3.4**

AST length for each dataset into Eq. (8), α is 1.96 for GEO, 1.27 for ATIS, 1.75 for DJANGO, and 1.23 for CONALA, which are all in the range of 1 to 2. Therefore, it is expected that the model works well when α is equal to 1 and 2. These results demonstrate the advantage of using Eq. (8) to calculate α automatically when scaling factor f is continuous. Note that Eq. (6) is not continuous, therefore we intuitively choose the α equal to 2, which has shown excellent performance in each dataset.

6.2 Computational Overhead

Regarding the computational overhead of AP Loss, theoretically, the computation of AST2Vec is required with additional computational overhead during the training stage. However, in the data preparation stage, AST2Vec can be computed and stored while building the AST of the training data. This allows direct utilization of AST2Vec during the actual training stage, minimizing additional computational load. Furthermore, in the inference stage, our method incurs no additional computational overhead. Therefore, the additional computational overhead brought by the AP loss is very small.

7 Related Work

Recently, lots of code generation models have achieved excellent success [9, 32, 34, 6, 31, 22, 4, 13, 5, 42, 17]. The work [19] treated code generation as conditional text generation and solved it with the sequence-to-sequence model. In order to utilize the grammatical information of the code, the Seq2Tree models are resorted, transforming NL utterances into a sequence of AST-based grammar actions. Typically, Dong et al. [2] first explored a Seq2Tree model for code generation. The work [25] studied a modular encoder-decoder architecture with structured AST output spaces. Yin et al. [39] proposed a Seq2Tree model to generate AST as the intermediate representation of codes. Moreover, the work [29] used the Transformer [30] architecture and attention mechanisms to address the long dependency problem for code generation. The authors in [11] explored the use of the future nodes information generated by AST for prediction. The work [26] proposed Subtoken-TranX adopted by Alibaba's BizCook platform, which is the first domain code generation system adopted in industrial development environments.

Some researchers have also noticed that the importance of each AST node for code generation is different. Xie et al. [33] proposed a mutual learning framework for Seq2Tree models to learn the information of AST nodes in different traversals simultaneously. The work [12] selected the generation path of the node in AST using reinforcement learning. Significantly different from the above work, we explore the impact of the position information of AST nodes during the inference stage and propose an effective method to focus on the more critical antecedent nodes during the training stage.

8 Threats to Validity

There are three major threats to the validity of our work.

1) **Threats to external validity** concern the quality of experimental datasets and the generalizability of our results. First, the four public datasets for code generation are mainstream benchmarks and have been used in many related works [39, 40, 33, 12, 11]. Second, AP Loss can be applied to any Seq2Tree model, and Seq2Seq models should further take into account the relationship between AST and code to employ AP Loss. In addition, APT uses only language-agnostic features and can be adapted to other programming languages.

2) **Threats to internal validity** involve the impact of hyperparameters. Deep learning models are known to be sensitive to hyperparameters. For the baselines, we work with the source code supplied by their original papers and ensure that the performance of the mode is comparable with their reported results. For our APT, we do a small-range grid search on hyperparameter γ, leaving other hyperparameters the same as those in previous studies [39, 40]. Previous work [39, 40] has explored effective settings of the hyperparameters through extensive experiments. Therefore, there may be room to tune more hyperparameters for additional improvements.

3) **Threats to construct validity** pertain to the reliability of our evaluation metrics. To address this threat, we employ EM and BLEU as evaluation metrics. EM evaluates the percentage of correctly predicted code snippets, and BLEU measures the text similarity between predictions and the ground truth. They are the mainstream metrics for code generation and are used in most previous studies [39, 40, 33, 12, 11]. Based on the above metrics, each experiment is run five times, and its average result is reported.

9 Conclusion and Future Work

In this paper, we have identified the prediction of antecedent nodes as a critical factor influencing the behavior of Seq2Tree models. To address this issue, we have proposed an effective method, namely AP Loss, that adjusts the standard cross entropy loss to prioritize learning on antecedent nodes while down-weight the more susceptible subsequent nodes. Additionally, we have proposed an AST2Vec method that models AST using vectors to differentiate between antecedent and subsequent nodes. Leveraging AP Loss, APT outperforms existing Seq2Tree methods and achieves SOTA performance across four benchmarks. Comprehensive experimental results and analyses substantiate the effectiveness and versatility of our proposed methods.

We demonstrate the efficacy of AP Loss by incorporating it into some widely-used Seq2Tree methods. However, the performance of APT is inherently constrained by the capabilities of the base model. In future work, we envisage applying AP Loss to training a larger Seq2Tree model following the method proposed by this paper, which could potentially lead to substantially superior outcomes. Furthermore, we utilize AST2Vec to weigh AST nodes for AP Loss in this paper, and we intend to investigate the application of AST2Vec as a positional encoding in future research.

10 Acknowledgments

References

[1] David Alvarez-Melis and Tommi S. Jaakkola, 'Tree-structured decoding with doubly-recurrent neural networks', in *ICLR*, (2017).

[2] Li Dong and Mirella Lapata, 'Language to logical form with neural attention', in *ACL*, pp. 33–43, (2016).

[3] Li Dong and Mirella Lapata, 'Coarse-to-fine decoding for neural semantic parsing', in *ACL*, pp. 731–742, (2018).

[4] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin, 'Codescore: Evaluating code generation by learning code execution', *CoRR*, **abs/2301.09043**, (2023).

[5] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li, 'Self-collaboration code generation via chatgpt', *CoRR*, **abs/2304.07590**, (2023).

[6] Yihong Dong, Ge Li, and Zhi Jin, 'CODEP: grammatical seq2seq model for general-purpose code generation', in *ISSTA*, pp. 188–198. ACM, (2023).

[7] Shuzheng Gao, Xinjie Wen, Cuiyun Gao, Wenxuan Wang, and Michael R. Lyu, 'Constructing effective in-context demonstration for code intelligence tasks: An empirical study', *ASE*, (2023).

[8] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin, 'Interpretation-based code summarization', in *ICPC*, pp. 113–124. IEEE, (2023).

[9] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig, 'Retrieval-based neural code generation', in *EMNLP*, (2018).

[10] Charles T. Hemphill, John J. Godfrey, and George R. Doddington, 'The ATIS spoken language systems pilot corpus', in *NAACL*, (1990).

[11] Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su, 'An AST structure enhanced decoder for code generation', *IEEE ACM Trans. Audio Speech Lang. Process.*, **30**, 468–476, (2022).

[12] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su, 'Exploring dynamic selection of branch expansion orders for code generation', in *ACL/IJCNLP*, pp. 5076–5085, (2021).

[13] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li, 'Self-planning code generation with large language model', *CoRR*, **abs/2303.06689**, (2023).

[14] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu, 'Treebert: A tree-based pre-trained model for programming language', in *UAI*, volume 161 of *Proceedings of Machine Learning Research*, pp. 54–63. AUAI Press, (2021).

[15] Diederik P. Kingma and Jimmy Ba, 'Adam: A method for stochastic optimization', in *ICLR*, (2015).

[16] Jia Li, Ge Li, Yongmin Li, and Zhi Jin, 'Enabling programming thinking in large language models toward code generation', *CoRR*, **abs/2305.06599**, (2023).

[17] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu, 'Codeeditor: Learning to edit source code with pre-trained models', *ACM Trans. Softw. Eng. Methodol.*, (2023).

[18] Jia Li, Yuyuan Zhao, Zhi Jin, Ge Li, Tao Shen, Zhengwei Tao, and Chongyang Tao, 'SK2: integrating implicit sentiment knowledge and explicit syntax knowledge for aspect-based sentiment analysis', in *CIKM*, pp. 1114–1123. ACM, (2022).

[19] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior, 'Latent predictor networks for code generation', in *ACL*, pp. 599–609, (2016).

[20] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin, 'Multi-task learning based pre-trained language model for code completion', in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 473–485, (2020).

[21] Fang Liu, Jia Li, and Li Zhang, 'Syntax and domain aware model for unsupervised program translation', *arXiv preprint arXiv:2302.03908*, (2023).

[22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu, 'Codexglue: A machine learning benchmark dataset for code understanding and generation', in *NeurIPS Datasets and Benchmarks*, (2021).

[23] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura, 'Learning to generate pseudo-code from source code using statistical machine translation (T)', in *ASE*, pp. 574–584. IEEE Computer Society, (2015).

[24] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu, 'Revisiting, benchmarking and exploring api recommendation: How far are we?', *IEEE Transactions on Software Engineering*, (2021).

[25] Maxim Rabinovich, Mitchell Stern, and Dan Klein, 'Abstract syntax networks for code generation and semantic parsing', in *ACL*, pp. 1139–1149, (2017).

[26] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li, 'Incorporating domain knowledge through task augmentation for front-end javascript code generation', in *ESEC/SIGSOFT FSE*, pp. 1533–1543. ACM, (2022).

[27] Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov, 'Program synthesis and semantic parsing with learned code idioms', in *NeurIPS*, pp. 10824–10834, (2019).

[28] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang, 'A grammar-based structural CNN decoder for code generation', in *AAAI*, pp. 7055–7062, (2019).

[29] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang, 'Treegen: A tree-based transformer architecture for code generation', in *AAAI*, pp. 8984–8991, (2020).

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin, 'Attention is all you need', in *NeurIPS*, pp. 5998–6008, (2017).

[31] Yue Wang, Weishi Wang, Shafiq R. Joty, and teven C. H. Hoi, 'Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation', in *EMNLP*, pp. 8696–8708, (2021).

[32] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin, 'Code generation as a dual task of code summarization', in *NeurIPS*, pp. 6559–6569, (2019).

[33] Binbin Xie, Jinsong Su, Yubin Ge, Xiang Li, Jianwei Cui, Junfeng Yao, and Bin Wang, 'Improving tree-structured decoder training for code generation via mutual learning', in *AAAI*, pp. 14121–14128, (2021).

[34] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig, 'Incorporating external knowledge through pre-training for natural language to code generation', in *ACL*, pp. 6045–6052, (2020).

[35] Zhou Yang, Jieke Shi, Junda He, and David Lo, 'Natural attack for pre-trained models of code', in *ICSE*, pp. 1482–1493. ACM, (2022).

[36] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo, 'Stealthy backdoor attack for code models', *CoRR*, **abs/2301.02496**, (2023).

[37] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig, 'Learning to mine aligned code and natural language pairs from stack overflow', in *MSR*, pp. 476–486, (2018).

[38] Pengcheng Yin and Graham Neubig, 'A syntactic neural model for general-purpose code generation', in *ACL*, pp. 440–450, (2017).

[39] Pengcheng Yin and Graham Neubig, 'TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation', in *EMNLP*, pp. 7–12, (2018).

[40] Pengcheng Yin and Graham Neubig, 'Reranking for neural semantic parsing', in *ACL*, pp. 4553–4559, (2019).

[41] John M. Zelle and Raymond J. Mooney, 'Learning to parse database queries using inductive logic programming', in *AAAI*, pp. 1050–1055, (1996).

[42] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin, 'Self-edit: Fault-aware code editor for code generation', in *ACL (1)*, pp. 769–787. Association for Computational Linguistics, (2023).

[43] Yunfei Zhao, Yihong Dong, and Ge Li, 'Seq2seq or seq2tree: Generating code using both paradigms via mutual learning', in *Internetware*. ACM, (2023).