TAG : Type Auxiliary Guiding for Code Comment Generation

Ruichu Cai[1], **Zhihao Liang**[1], **Boyan Xu**[1*], **Zijian Li**[1], **Yuexing Hao**[2], **Yao Chen**[3]
[1] School of Computer Science, Guangdong University of Technology, China
[2] Rutgers University New Brunswick, USA
[3] Advanced Digital Sciences Center, Singapore
cairuichu@gmail.com, zhihaolzh95@gmail.com, hpakyim@gmail.com,
leizigin@gmail.com, yh599@scarletmail.rutgers.edu, yao.chen@adsc-create.edu.sg

Abstract

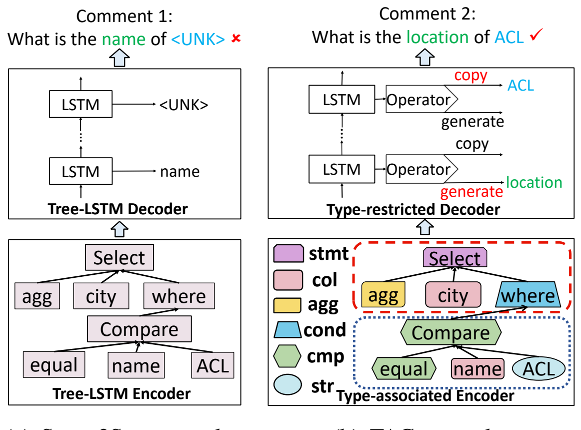
Existing leading code comment generation approaches with the structure-to-sequence framework ignores the type information of the interpretation of the code, e.g., operator, string, etc. However, introducing the type information into the existing framework is non-trivial due to the hierarchical dependence among the type information. In order to address the issues above, we propose a Type Auxiliary Guiding encoder-decoder framework for the code comment generation task which considers the source code as an N-ary tree with type information associated with each node. Specifically, our framework is featured with a *Type-associated Encoder* and a *Type-restricted Decoder* which enables adaptive summarization of the source code. We further propose a hierarchical reinforcement learning method to resolve the training difficulties of our proposed framework. Extensive evaluations demonstrate the state-of-the-art performance of our framework with both the auto-evaluated metrics and case studies.

1 Introduction

The comment for the programming code is critical for software development, which is crucial to the further maintenance of the project codebase with significant improvement of the readability (Aggarwal et al., 2002; Tenny, 1988). Code comment generation aims to automatically transform program code into natural language with the help of deep learning technologies to boost the efficiency of the code development.

Existing leading approaches address the code comment generation task under the structure-to-sequence (Struct2Seq) framework with an encoder-decoder manner by taking advantage of the inherent structural properties of the code. For instance, existing solutions leverage the syntactic structure of abstract syntax trees (AST) or parse trees from

[*] Corresponding author



(a) Struct2Seq example (b) TAG example

Figure 1: Comment generation frameworks. Different types are denoted as different colors and shapes in (b).

source code have shown significant improvement to the quality of the generated comments (Liang and Zhu, 2018; Alon et al., 2018; Hu et al., 2018; Wan et al., 2018); Solutions representing source code as graphs have also shown high-quality comment generation abilities by taking advantage of extracting the structural information of the codes (Xu et al., 2018a,b; Fernandes et al., 2018).

Although promising results were reported, we observe that the information of the node type in the code is not considered in these aforementioned Struct2Seq based solutions. The lack of such essential information lead to the following common limitations: 1) Losing the accuracy for encoding the source code with the same structure but has different types. As shown in Fig. 1(a), a Tree-LSTM (Tai et al., 2015) encoder is illustrated to extract the structural information, the two subtrees of the code 'Select' and 'Compare' in the dashed box have the same structure but different types, with the ignorance of the type information, the traditional encoders illustrate the same set of neural network parameters to encode the tree, which leads to an inaccurate generation of the comment. 2) Losing both the efficiency and accuracy for searching the large vocabulary in the decoding procedure, especially for the out-of-vocabulary (OOV) words that exist in the source code but not in the target dictionary. As shown in the Fig. 1(a), missing the type of 'ACL' node usually results in an unknown word 'UNK' in the generated comments. Thus, the key to tackle these limitations is efficiently utilizing the node type information in the encoder-decoder framework.

To well utilize the type information, we propose a Type Auxiliary Guiding (TAG) encoder-decoder framework. As shown in Fig. 1(b), in the encoding phase, we devise a *Type-associated encoder* to encode the type information in the encoding of the N-ary tree. In the decoding phase, we facilitate the generation of the comments with the help of type information in a two-stage process naming *operation selection* and *word selection* to reduce the searching space for the comment output and avoid the out-of-vocabulary situation. Considering that there is no ground-truth labels for the operation selection results in the two-stage generation process, we further devised a Hierarchical Reinforcement Learning (HRL) method to resolve the training of our framework. Our proposed framework makes the following contributions:

- An adaptive *Type-associated encoder* which can summarize the information according to the node type;
- A *Type-restricted decoder* with a two-stage process to reduce the search space for the code comment generation;
- A hierarchical reinforcement learning approach that jointly optimizes the operation selection and word selection stages.

2 Related Work

Code comment generation frameworks generate natural language from source code snippets, e.g. SQL, lambda-calculus expression and other programming languages. As a specified natural language generation task, the mainstream approaches could be categorized into textual based method and structure-based method.

The textual-based method is the most straightforward solution which only considers the sequential text information of the source code. For instance, Movshovitz-Attias and Cohen (2013) uses topic models and n-grams to predict comments with given source code snippets; Iyer et al. (2016) presents a language model Code-NN using LSTM networks with attention to generate descriptions about C# and SQL; Allamanis et al. (2016) predicts summarization of code snippets using a convolutional attention network; Wong and Mooney (2007) presents a learning system to generate sentences from lambda-calculus expressions by inverting semantic parser into statistical machine translation methods.

The structure-based methods take the structure information into consideration and outperform the textual-based methods. Alon et al. (2018) processes a code snippet into the set of compositional paths in its AST and uses attention mechanism to select the relevant paths during the decoding. Hu et al. (2018) presents a Neural Machine Translation based model which takes AST node sequences as input and captures the structure and semantic of Java codes. Wan et al. (2018) combines the syntactic level representation with lexical level representation by adopting a tree-to-sequence (Eriguchi et al., 2016) based model. Xu et al. (2018b) considers a SQL query as a directed graph and adopts a graph-to-sequence model to encode the global structure information.

Copying mechanism is utilized to address the OOV issues in the natural language generation tasks by reusing parts of the inputs instead of selecting words from the target vocabulary. See et al. (2017) presents a hybrid pointer-generator network by introducing pointer network (Vinyals et al., 2015) into a standard sequence-to-sequence (Seq2Seq) model for abstractive text summarization. COPYNET from Gu et al. (2016) incorporates the conventional copying mechanism into Seq2Seq model and selectively copy input segments to the output sequence. In addition, Ling et al. (2016) uses the copying mechanism to copy strings from the code.

Our targeted task is considered as the opposite process of natural language to programming code (NL-to-code) task. So some of the NL-to-code solutions are also taken as our references. Dong and Lapata (2016) distinguishes types of nodes in the logical form by whether nodes have child nodes. Yin and Neubig (2017); Rabinovich et al. (2017); Xu et al. (2018a) take the types of AST nodes into account and generate the corresponding programming codes. Cai et al. (2018) borrows the idea of Automata theory and considers the specific types of SQL grammar in Backus-Naur form (BNF) and generates accurate SQL queries with the help of it.

Inspired by the methods considering the type

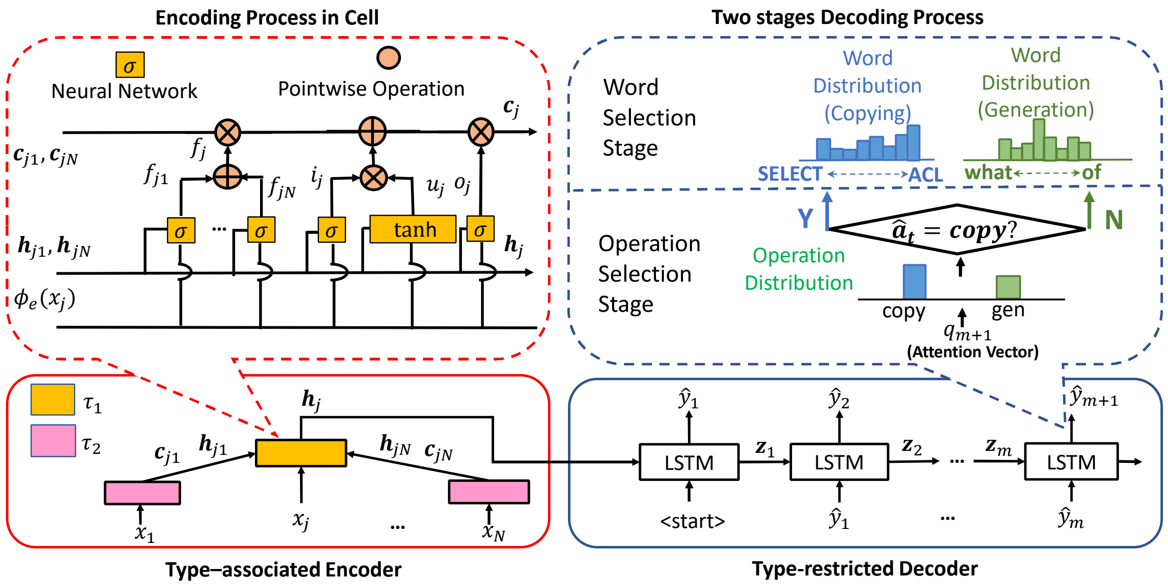


Figure 2: TAG Encoder and Decoder framework.

information of the code, our solution differs from the existing method with a *Type-associated Encoder* that encodes the type information during the substructure summarization and a *Type-restricted Decoder* that can reduce search space for the code comment generation. In addition, two improvements are developed according to our objectives. First, we design a type-restricted copying mechanism to reduce the difficulty of extracting complex grammar structure from the source code. Second, we use a hierarchical reinforcement learning methods to train the model in our framework to learn to select from either copy or other actions, the details will be presented in Section 3.

3 Model Overview

We first make the necessary definition and formulation for the input data and the code comment generation problem for our Type Auxiliary Guiding (TAG) encoder-decoder framework.

Definition 1 ***Token-type-tree.*** *Token-type-tree $T_{x,\tau}$ represents the source code with the node set V, which is a rooted N-ary tree. And $V = \{v_1, v_2, .., v_{|V|}\}$ denotes a partial order nodes set satisfying $v_1 \preceq v_2 \preceq ..., \preceq v_{|V|}$. Let internal node $v_j = \{x_j, \tau_j\}$, where x_j denotes the token sequence and τ_j denotes a type from grammar type set $\mathcal{T}$.*

Token-type-tree can be easily constructed from token information of the original source code and type information of its AST or parse tree. According to Definition 1, we formulate the code comment generation task as follows.

Formulation 1 ***Code Comment Generation with Token-type-tree as the Input.*** *Let $\mathcal{S}$ denote training dataset and labeled sample $(T_{x,\tau}, \boldsymbol{y}) \in \mathcal{S}$, where $T_{x,\tau}$ is the input token-type-tree, $\boldsymbol{y} = (y_1, y_2, \cdots, y_M)$ is the ground truth comment with M words. The task of code comment generation is to design a model which takes the unlabeled sample $T_{x,\tau}$ as input and predicts the output as its comment, denoted as $\boldsymbol{y}$.*

Our framework follows the encoder-decoder manner, and consists of the revised two major components, namely the *Type-associated Encoder* and *Type-restricted Decoder*. As shown in Fig. 2.

The *Type-associated Encoder*, as shown in Fig. 2, recursively takes the token-type-tree $T_{x,\tau}$ as input, and maintains the semantic information of the source code in the hidden states. Instead of using the same parameter sets to learn the whole token-type-tree, *Type-associated Encoder* utilizes multiple sets of parameters to learn the different type of nodes. The parameters of the cells are adaptively invoked according to the type of the current node during the processing of the input token-type-tree. Such a procedure enables the structured semantic representation to contain the type information of the source code.

The *Type-restricted Decoder*, as shown in the right part of Figure 2, takes the original toke-type-tree $T_{x,\tau}$ and its semantic representation from encoder as input and generates the corresponding comment. Different from conventional decoders which generate output only based on the target dictionary, our *Type-restricted Decoder* considers both input code to the encoder and target dictionary as the source of output. Attention mechanism is employed to compute an attention vector which is used to generate the output words through a two-stage process: (1) Determine either to copy from the original token-type-tree or to generate from the current hidden state according to the distribution of the operation. (2) If the copying operation is selected, the words are copied from the selected node from the token-type-tree $T_{x,\tau}$ with restricted types; otherwise, the candidate word will be selected from the target dictionary. The above two-stage process is guided by the type which is extracted from the hidden state of encoder with the help of attention mechanism. Such a process enables adaptive switching between copying and generation processes, and not only reduces the search space of the generation process but also addresses the OOV problem with the copying mechanism.

Although the proposed framework provides an efficient solution with the utilization of the type information in the code, training obstacles are raised accordingly: (1) No training labels are provided for the operation selection stage. (2) There is a mismatch between the evaluation metric and the objective function. Thus, we further devised an HRL method to train our TAG model. In the HRL training, the TAG model feeds back the evaluation metric as the learning reward to train the two-stage sampling process without relying on the ground-truth label of operation selection stage.

4 Type-associated Encoder

The encoder network aims to learn a semantic representation of the input source code. The key challenge is to provide distinct summarization for the sub-trees with the same structure but different semantics. As shown in the Type-associated Encoder in Fig. 1, the blue and red dashed blocks have the same 3-ary substructure. The sub-tree in the blue box shares the same sub-structure with the tree in the red box, which is usually falsely processed by the same cell in a vanilla Tree-LSTM. By introducing the type information, the semantics of the two subtrees are distinguished from each other.

Our proposed Type-associated Encoder is designed as a variant N-ary Tree-LSTM. Instead of directly inputting type information as features into the encoder for learning, we integrate the type information as the index of the learning parameter sets of the encoder network. More specifically, different sets of parameters are defined through different types, which provides a more detailed summarization of the input. As is shown in Fig. 1(b), the two sub-trees in our proposed Type-associated Encoder are distinguished by the type information. The tree contains N ordered child nodes, which are indexed from 1 to N. For the j-th node, the hidden state and memory cell of its k-th child node is denoted as $\boldsymbol{h}_{jk}$ and $\boldsymbol{c}_{jk}$, respectively. In order to effectively capture the type information, we set $\boldsymbol{W}_{\tau_j}$ and $\boldsymbol{b}_{\tau_j}$ to be the weight and bias of the j-th node, and $\boldsymbol{U}_{\tau_{jk}}$ be the weight of the k-th child of the j-th node. The transition equation of the variant N-ary Tree-LSTM is shown as follow:

$$\boldsymbol{i}_j = \sigma\left(\boldsymbol{W}^{(\boldsymbol{i})}_{\tau_j}\phi\left(x_j\right) + \sum_{l=1}^{N}\boldsymbol{U}^{(\boldsymbol{i})}_{\tau_{jl}}\boldsymbol{h}_{jl} + \boldsymbol{b}^{(\boldsymbol{i})}_{\tau_j}\right), \quad (1)$$

$$\boldsymbol{f}_{jk} = \sigma\left(\boldsymbol{W}^{(\boldsymbol{f})}_{\tau_{jk}}\phi\left(x_j\right) + \sum_{l=1}^{N}\boldsymbol{U}^{(\boldsymbol{f})}_{\tau_{jl,k}}\boldsymbol{h}_{jl} + \boldsymbol{b}^{(\boldsymbol{f})}_{\tau_{jk}}\right), \quad (2)$$

$$\boldsymbol{o}_j = \sigma\left(\boldsymbol{W}^{(\boldsymbol{o})}_{\tau_j}\phi\left(x_j\right) + \sum_{l=1}^{N}\boldsymbol{U}^{(\boldsymbol{o})}_{\tau_{jl}}h_{jl} + \boldsymbol{b}^{(\boldsymbol{o})}_{\tau_j}\right), \quad (3)$$

$$\boldsymbol{u}_j = \tanh\left(\boldsymbol{W}^{(\boldsymbol{u})}_{\tau_j}\phi\left(x_j\right) + \sum_{l=1}^{N}\boldsymbol{U}^{(\boldsymbol{u})}_{\tau_{jl}}h_{jl} + \boldsymbol{b}^{(\boldsymbol{u})}_{\tau_j}\right), \quad (4)$$

$$\boldsymbol{c}_j = \boldsymbol{i}_j \odot \boldsymbol{u}_j + \sum_{l=1}^{N}\boldsymbol{f}_{jl} \odot \boldsymbol{c}_{jl}, \quad (5)$$

$$\boldsymbol{h}_j = \boldsymbol{o}_j \odot \tanh\left(\boldsymbol{c}_j\right), \quad (6)$$

We employ the forget gate (Tai et al., 2015) for the Tree-LSTM, the parameters for the k-th child of the j-th node's is denoted as $\boldsymbol{f}_{jk}$. $\boldsymbol{U}_{\tau_{jl,k}}$ is used to represent the weight of the type for the l-th child of the j-th node in the k-th forget gate. The major difference between our variants and the traditional Tree-LSTM is that the parameter set ($\boldsymbol{W}_\tau$, $\boldsymbol{U}_\tau$, $\boldsymbol{b}_\tau$) are specified for each type τ.

5 Type-restricted Decoder

Following with the Type-associated Encoder, we propose a Type-restricted Decoder for the decoding phase, which incorporates the type information into its two-stage generation process. First of all, an attention mechanism is adopted in the decoding phase which takes hidden states from the encoder as input and generates the attention vector. The resulted attention vector is used as input to the following two-stage process, named *operation selection stage* and *word selection stage*, respectively. The operation selection stage selects between generation operation and copying operation for the following word selection stage. If the generation operation is selected, the predicted word will be generated from the targeted dictionary. If the copying operation is selected, then a type-restricted copying mechanism is enabled to restrict the search space by masking down the illegal grammar types. Furthermore, a copying decay strategy is illustrated to solve the issue of repetitively focusing on specific nodes caused by the attention mechanism. The details of each part are given below.

Attention Mechanism: The encoder extracts the semantic representation as the hidden state of the rooted nodes, denoted as $\boldsymbol{h}_r$, which are used to initialize the hidden state of the decoder, $\boldsymbol{z}_0 \leftarrow \boldsymbol{h}_r$. At time step m, given output y_{m-1} and the hidden state of the decoder $\boldsymbol{z}_{m-1}$ at last time step $m-1$, the hidden state $\boldsymbol{z}_m$ is recursively calculated by the LSTM cells in the decoder,

$$\boldsymbol{z}_m = LSTM(\boldsymbol{z}_{m-1}, y_{m-1}). \tag{7}$$

The attention vector $\boldsymbol{q}$ is calculate with:

$$\begin{aligned} \alpha_{mj} &= \frac{\exp\left(\boldsymbol{h}_j^\top \boldsymbol{z}_m\right)}{\sum_{j=1}^{|V_x|} \exp\left(\boldsymbol{h}_j^\top \boldsymbol{z}_m\right)}, \\ \widetilde{\boldsymbol{q}_m} &= \sum_{j=1}^{|V_x|} \alpha_{mj} \boldsymbol{h}_j, \\ \boldsymbol{q}_m &= \tanh\left(\boldsymbol{W}_q \left[\widetilde{\boldsymbol{q}}, \boldsymbol{z}_m\right]\right), \end{aligned} \tag{8}$$

where $\boldsymbol{W}_q$ is the parameters of the attention mechanism. The attention vector contains the token and type information, which is further facilitated in the following operation selection and word selection stages.

Operation Selection Stage: Operation Selection Stage determines either using the copying operation or the generation operation to select the words based on the attention vector and hidden states from the encoder. Specifically, given the attention vector $\boldsymbol{q}_m$ at time step m, Operation Selection Stage estimates the conditional probabilities as the distribution of the operation $p(\hat{a}_m|\hat{y}_{<m}; T_{x,\tau})$, where $\hat{a}_m \in \{0, 1\}$ and 0 and 1 represents the copy and the generation operations, respectively. A fully connected layer followed by a softmax is implemented to compute the distribution of the operations.

$$p(\hat{a}_m|\hat{y}_{<m}; T_{x,\tau}) = softmax(\boldsymbol{W}_s \boldsymbol{q}_m), \tag{9}$$

The $\boldsymbol{W_s}$ in the Eq. 9 is the trainable parameters. Since there is no ground-truth label for operation selection, we employ an HRL method to jointly train the operation selection stage and the following stage, the details are provided in Section 6.

Word Selection Stage: Word Selection Stage also contains two branches. The selection between them is determined by the previous stage. If the generation operation is selected in the Operation Selectoin Stage, the attention vector will be fed into a softmax layer to predict the distribution of the target word, formulated as

$$p(y_m|\hat{a}_m = 1, \hat{y}_{<m}; T_{x,\tau}) = softmax\left(\boldsymbol{W}_g \boldsymbol{q}_m\right), \tag{10}$$

where $\boldsymbol{W}_g$ is the trainable parameters of the output layer. Otherwise, if the copy operation is selected, we employ the dot-product score function to calculate score vector $\boldsymbol{s}_m$ of the hidden state of the node and the attention vector. Similarly, score vector $\boldsymbol{s}_m$ will be fed into a softmax layer to predict the distribution of the input word, noted as:

$$\begin{aligned} \boldsymbol{s}_m &= \left[\boldsymbol{h}_1, \boldsymbol{h}_2, \cdots, \boldsymbol{h}_{|V_x|}\right]^\top \boldsymbol{q}_m \\ p(y_m|\hat{a}_m = 0; \hat{y}_{<m}; T_{x,\tau}) &= softmax\left(\boldsymbol{s}_m\right). \end{aligned} \tag{11}$$

One step further, to filter out the illegally copied candidates, we involve a grammar-type based mask vector $\boldsymbol{d}_m \in \mathbb{R}^{|V_x|}$ at each decoding step m. Each dimension of $\boldsymbol{d}_m$ corresponds to each node of the token-type-tree. If the mask of the node in token-type-tree indicates the node should be filtered out, then the corresponding dimension is set as negative infinite. Otherwise, it is set to 0. Thus, the restricted copying stage is formulated as

$$p(y_m|\hat{a}_m = 0, \hat{y}_{<m}; T_{x,\tau}) = softmax\left(\boldsymbol{s}_m + \boldsymbol{d}_m\right). \tag{12}$$

The word distribution of the two branches is represented with a softmax over input words or target dictionary words in Eq. 10 and Eq. 12. At each time step, the word with the highest probability in the word distribution will be selected.

Copying Decay Strategy: Similar to the conventional copying mechanism, we also use the attention vector as a pointer to guide the copying process. The type-restricted copying mechanism tends to pay more attention to specific nodes, resulting in the ignorance of other available nodes, which makes certain copied tokens repeatedly active in a short distance in a single generated text, lead to a great redundancy of the content.

So we design a *Copying Decay Strategy* to smoothly penalize certain probabilities of outstandingly copied nodes. We define a copy time-based decay rate λ_{mi} for the i-th tree node x_i in the m-th decoding step. If one node is copied in time step m, its decay rate is initialized as 1. In the next time step $m+1$, it is scaled by a coefficient $\gamma \in (0,1)$:

$$\lambda_{m+1,i} = \gamma\lambda_{m,i} \tag{13}$$

The overall formulation for the Type-restricted Decoder is:

$$\begin{aligned} p(y_m|\hat{a}_m = 0, \hat{y}_{<m}; T_{x,\tau}) = \\ softmax\,(\boldsymbol{s}_m + \boldsymbol{d}_m) \odot (1 - \boldsymbol{\lambda}_m) \end{aligned} \tag{14}$$

6 Hierarchical Reinforcement Learning

There remain two challenges to train our proposed framework, which are 1) the lack of ground truth label for the operation selection stage and 2) the mismatch between the evaluation metric and objective function. Although it is possible to train our framework by using the maximum likelihood estimation (MLE) method which constructs pseudo-labels or marginalize all the operations in the operation selection stage (Jia and Liang, 2016; Gu et al., 2016), the loss-evaluation mismatch between MLE loss for training and non-differentiable evaluation metrics for testing lead to inconsistent results (Keneshloo et al., 2019; Ranzato et al., 2015). To address these issues, we propose a Hierarchical Reinforcement Learning method to train the operation selection stage and word selection stage jointly.

We set the objective of the HRL as maximizing the expectation of the reward $R(\hat{\boldsymbol{y}}, \boldsymbol{y})$ between the predicted sequence $\hat{\boldsymbol{y}}$ and the ground-truth sequence $\boldsymbol{y}$, denoted as L_r. It could be formulated as a function of the input tuple $\{T_{x,\tau}, \boldsymbol{y}\}$ as,

$$\begin{aligned} L_r &= \frac{1}{|\mathcal{S}|}\sum_{(T_{x,\tau},\boldsymbol{y})\in\mathcal{S}} E_{\hat{\boldsymbol{y}}\sim p(\hat{\boldsymbol{y}}|T_{x,\tau})}[R(\hat{\boldsymbol{y}},\boldsymbol{y})] \\ &= \frac{1}{|\mathcal{S}|}\sum_{(T_{x,\tau},\boldsymbol{y})\in\mathcal{S}}\sum_{\hat{\boldsymbol{y}}\in Y} p(\hat{\boldsymbol{y}}|T_{x,\tau})R(\hat{\boldsymbol{y}},\boldsymbol{y}), \end{aligned} \tag{15}$$

Here, Y is the set of the candidate comment sequences. The reward $R(\hat{(\boldsymbol{y})}, \boldsymbol{y})$ is the non-differentiable evaluation metric, i.e., BLEU and ROUGE (details are in Section 7). The expectation in Eq. (15) is approximated via sampling $\hat{\boldsymbol{y}}$ from the distribution $p(\hat{\boldsymbol{y}}|T_{x,\tau})$. The procedure of sampling $\hat{\boldsymbol{y}}$ from $p(\hat{\boldsymbol{y}}|T_{x,\tau})$ is composed of the sub-procedures of sampling $\hat{y}_m$ from $p(\hat{y}_m|\hat{y}_{<m}; T_{x,\tau})$ in each decoding step m.

As mentioned above, the predicted sequence $\hat{\boldsymbol{y}}$ comes from the two branches of Word Selection Stage, depending on the Operation Selection Stage. a is defined as the action of the Operation selection stage. After involving the action a_m in time step m, Eq. (15) can be constructed by the joint distribution of the two stages:

$$\begin{aligned} &\frac{1}{|\mathcal{S}|}\sum_{(T_{x,\tau},\boldsymbol{y})\in\mathcal{S}}\sum_{\hat{\boldsymbol{y}}\in Y} p(\hat{\boldsymbol{y}}|T_{x,\tau})R(\hat{\boldsymbol{y}},\boldsymbol{y}) \\ &= \frac{1}{|\mathcal{S}|}\sum_{\dots}\sum_{\hat{\boldsymbol{y}}\in Y}\Big(\prod_{m=1}^{M}\sum_{\hat{a}_m}\underbrace{p(\hat{y}_m, \hat{a}_m|\hat{y}_{<m}; T_{x,\tau})}_{\text{Two-stage Joint Distribution}}\Big)R(\hat{\boldsymbol{y}},\boldsymbol{y}) \\ &= \dots\underbrace{p(\hat{y}_m|\hat{a}_m;\hat{y}_{<m};T_{x,\tau})}_{\text{Word Distribution}}\underbrace{p(\hat{a}_m|\hat{y}_{<m};T_{x,\tau})}_{\text{Operation Distribution}}\dots \end{aligned} \tag{16}$$

As shown in Eq. (16), the model finally selects the word $\hat{y}_m$ in time step m from the word distribution conditioned on $\hat{y}_{<m}$, $T_{x,\tau}$ and the operation $\hat{a}_m$ which is determined in the operation selection stage. In other words, there is a hierarchical dependency between the word selection stage and the operation selection stage.

As mentioned above, Y represents the space for all candidate comments, which is too large to practically maximize L_r. Since decoding is constructed via sampling from $p(\hat{y}_m|\hat{a}_m, \hat{y}_{<m}; T_{x,\tau})$ and $p(\hat{a}_m|\hat{y}_{<m}; T_{x,\tau})$, We adopt the Gumbel-Max solution (Gumbel, 1954) for the following sampling procedure:

$$\begin{aligned} \hat{a}_m &\sim p(\hat{a}_m|\hat{y}_{<m}; T_{x,\tau}), \\ \hat{y}_m &\sim p(\hat{y}_m|\hat{a}_m, \hat{y}_{<m}; T_{x,\tau}). \end{aligned} \tag{17}$$

Through the maximum sampling step M, Eq. (16) could be further approximated as the following equation:

$$\hat{L}_r = \frac{1}{|\mathcal{S}|}\sum_{\boldsymbol{y}\in\mathcal{S}} R(\hat{\boldsymbol{y}},\boldsymbol{y}) \tag{18}$$

The objective in Eq. (18) remains another challenge: for the entire sequence $\hat{\boldsymbol{y}}$, there is only a final reward $R(\hat{\boldsymbol{y}},\boldsymbol{y})$ available for model training, which is a sparse reward and leads to inefficient training of the model. So we introduce reward shaping (Ng et al., 1999) strategy to provide intermediate rewards to proceed towards the training goal, which adopts the accumulation of the intermediate rewards to update the model.

To further stabilize the HRL training process, we combine our HRL objective with the maximum-likelihood estimation(MLE) function according to

Wu et al. (2018a, 2016); Li et al. (2017); Wu et al. (2018b):

$$L_e = \frac{1}{|\mathcal{S}|} \sum_{(T_{x,\tau}, \boldsymbol{y}) \in \mathcal{S}} \sum_{\hat{\boldsymbol{y}} \in Y} \log p(\boldsymbol{y}|T_{x,\tau}) \tag{19}$$

$$L = \mu L_e + (1-\mu)\hat{L}_r,$$

where μ is a variational controlling factor that controls the trade-off between maximum-likelihood estimation function and our HRL objective. In the current training step tr, μ varies according to the training step tt as follows:

$$\mu = 1 - \frac{tr}{tt} \tag{20}$$

7 Evaluation and Analysis

7.1 Experimental Setup

7.1.1 Datasets

We evaluate our TAG framework on three widely used benchmark data sets, which are WikiSQL (Zhong et al., 2017), ATIS (Dong and Lapata, 2016) and CoNaLa (Yin et al., 2018). WikiSQL is a dataset of 80654 hand-annotated examples of SQL query and natural language comment pairs distributed across 24241 tables from Wikipedia. These SQL queries are further split into training (56355 examples), development (8421 examples) and test (15878 examples) sets. ATIS is in the form of lambda-calculus, which is a set of 5410 inquiries for flight information containing 4434 training examples, 491 development examples and 448 test examples. CoNaLa is a python related dataset. Its original version is used which includes 2879 snippet/intent pairs crawled from Stack Overflow, split into 2379 training and 500 test examples. We extract 200 random examples from its training set as the development set.

We transfer the SQL queries of WikiSQL into ASTs with 6 types according to the Abstract Syntax Description Language (ASDL) grammar, where the ASDL grammar for SQL queries is proposed in Yin and Neubig (2017). We transfer the lambda-calculus logical forms of ATIS to tree structure with 7 types according to the method proposed in Dong and Lapata (2016). The python snippets of CoNaLa are transformed into ASTs with 20 types, following the official ASDL grammar of python[1]. The data of the ASTs of these datasets is shown in Table 1, where the maximum depth of ASTs (Max-Tree-Depth), the maximum number of child nodes in ASTs (Max-Child-Count) and the average number of tree nodes in ASTs (Avg-Tree-Node-Count) are shown.

[1]https://docs.python.org/3.5/library/ast.html

Dataset	WikiSQL	ATIS	CoNaLa
Max Tree Depth	5	18	28
Max Child Num	4	15	10
Avg Tree Node Count	11.11	33.54	28.37

Table 1: Statistics of ASTs on the datasets.

7.1.2 Baselines Frameworks

We choose the representative designs for code comment generation as our baselines for comparison. Code-NN (Iyer et al., 2016) is chosen because of it is the first model to transform the source code into sentences. Pointer Generator (See et al., 2017) (P-G) is a seq2seq based model with a standard copying mechanism. In addition, we choose the attention based Tree-to-Sequence (Tree2Seq) model proposed by Eriguchi et al. (2016). Moreover, we also add the copying mechanism into Tree2Seq model as another baseline (T2S+CP). We choose Graph-to-Sequence (Graph2Seq) (Xu et al., 2018b) as a graph-based baseline for comparison. Since the authors have not released the code for data-preprocessing, we convert the tree-structured representation for the source code of SQL data into directed graphs for our replication.

7.1.3 Hyperparameters

Code-NN uses embedding size and hidden size both as 400, and applies random uniform initializer with 0.35 initialized weight, and adopts stochastic gradient descent algorithm to train the model with a learning rate at 0.5. P-G uses 128 embedding size, 256 hidden size and applies random uniform initializer with 0.02 initialized weights for initialization and Adam optimizer to train the model with 0.001 learning rate. Graph2Seq uses 100 embedding size, 200 hidden size and applies the truncated normal initializer for initialization. Adam optimizer is used to train the model with a 0.001 learning rate.

We use the Xavier initializer (Glorot and Bengio, 2010) to initialize the parameters of our proposed TAG framework. The size of embeddings is equivalent to the dimensions of LSTM states and hidden layers, which is 64 for ATIS and CoNaLa and 128 for WikiSQL. TAG is trained using the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.001. In order to reduce the size of the vocabulary, low-frequency words are not kept in both the

Model	WikiSQL (SQL)			ATIS (lambda-calculus)			CoNaLa (Python)		
	BLEU-4	ROUGE-2	ROUGE-L	BLEU-4	ROUGE-2	ROUGE-L	BLEU-4	ROUGE-2	ROUGE-L
Code-NN	6.7	9.7	30.9	37.1	43.28	59.4	8.1	12.2	26.1
P-G	25.7	29.2	50.1	41.9	47.3	60.5	10.0	13.8	28.0
Tree2Seq	22.0	22.0	43.4	40.1	47.2	60.9	6.6	9.2	25.2
Graph2Seq	17.6	24.3	45.7	34.6	41.8	58.3	10.4	14.1	28.2
T2S+CP	31.0	36.8	54.5	39.0	43.7	58.4	13.3	18.5	31.5
TAG(B)	**35.8**	41.0	57.8	**42.4**	**47.4**	61.2	**14.1**	19.4	31.8
TAG(R)	35.2	**41.1**	**58.1**	40.6	47.1	**61.5**	12.6	**19.7**	**32.2**

Table 2: Comparisons with baseline models on different test sets.

Model	BLEU-4	ROUGE-2	ROUGE-L
TAG-TA	34.8(-1.4)	41.0(-1.3)	57.8(-1.6)
TAG-MV	35.2(-1.0)	41.1(-1.2)	58.1(-1.3)
TAG-CD	33.5(-2.7)	40.0(-2.3)	57.1(-2.3)
TAG-RL	34.6(-1.6)	41.4(-0.9)	58.7(-0.7)
TAG(B)	**36.2**	42.0	58.8
TAG(R)	35.6	**42.3**	**59.4**

Table 3: Ablation study of TAG framework.

vocabulary for the source codes and the vocabulary for target comments. Specifically, the minimum threshold frequency for WikiSQL and ATIS is set as 4 while for CoNaLa it is set as 2. The hyperparameters of Tree2Seq and T2S+CP is equivalent to ours. The minibatch size of all the baseline models and ours are set to 32.

7.1.4 Evaluation Metric

We illustrate the n-gram based BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004) evaluations to evaluate the quality of our generated comments and also use them to set the reward in the HRL based training. Specifically, BLEU-4, ROUGE-2 and ROUGE-L are used to evaluate the performance of our model since they are the most representative evaluation metric for context-based text generation.

7.2 Results and Analysis

7.2.1 Comparison with the Baselines

Table 2 presents the evaluation results of the baseline frameworks and our proposed ones. Since our HRL could be switched to different reward functions, we evaluate both the BLEU oriented and ROUGE oriented training of our framework, denoted as TAG(B) and TAG(R). The results of TAG(B) and TAG(R) varies slightly compared to each other. However, both of them are significantly higher than all the selected counterparts, which demonstrates the state-of-the-art generation quality of our framework on all the datasets with different programming languages.

Specifically, TAG improves over 15% of BLEU-4, over 10% of ROUGE-2 and 6% of ROUGE-L on WikiSQL when compared to T2S+CP, which is the best one among all the baseline target for all the evaluations. For the lambda-calculus related corpus, TAG improves 1.0% of BLEU, 0.2% ROUGE-2 and 0.5% ROUGE-L on ATIS. The performance is more difficult to be improved on ATIS than the other two corpora due to the great dissimilarity of sub-trees of the lambda-calculus logical forms in it. In terms of the python related corpus, TAG improves 6% of BLEU, 6.4% of ROUGE-2 and 2.2% of ROUGE-L on CoNaLa when compared to the best one in our baselines. The low evaluation score and improvement of CoNaLa are due to the complex grammatical structures and lack of sufficient training samples, i.e., 20 types across only 2174 training samples, which result in an inadequately use of the advantage of our approach. However, our TAG framework still outperforms all the counterparts on these two datasets.

7.2.2 Ablation Study

To investigate the performance of each component in our model, we conduct ablation studies on the development sets. Since all the trends are the same, we omit the results on the other data sets and only present the ones of WikiSQL. The variants of our model are as follows:

- TAG-TA: remove *Type-associated Encoder*, use Tree-LSTM instead.
- TAG-MV: remove the mask vector $\boldsymbol{d}_m$.
- TAG-CD: remove Copying Decay Strategy.
- TAG-RL replace HRL with MLE, marginalize the actions of the operation selection.

The results of the ablation study are given in Table 3. Overall, all the components are necessary to TAG framework and providing important contributions to the final output. When compared to TAG-TA, the high performance of standard TAG

Code	Comment
SQL: SELECT MAX(Capacity) FROM table WHERE Stadium = “Otkrytie Arena”	**Ground-Truth**: What is the maximum capacity of the Otkrytie Arena Stadium ?
	Code-NN: What is the highest attendance for ?
	P-G: Who is the % that ’s position at 51 ?
	Tree2Seq: What is the highest capacity at <unk> at arena ?
	Graph2Seq: What is the highest capacity for arena arena ?
	T2S+CP: What is the highest capacity for the stadium ?
	TAG: What is the highest capacity for the stadium of Otkrytie Arena ?
Python: i: d [i] for i in d if i != ’c’	**Ground-Truth**: remove key ’c’ from dictionary ’d’
	Code-NN: remove all keys from a dictionary ’d’
	P-G: select a string ’c’ in have end of a list ’d’
	Tree2Seq: get a key ’key’ one ’,’ one ’,’ <unk>
	Graph2Seq: filter a dictionary of dictionaries from a dictionary ‘d’ where a dictionary of dictionaries ’d’
	T2S+CP: find all the values in dictionary ’d’ from a dictionary ’d’
	TAG: remove the key ’c’ if a dictionary ’d’

Table 4: Case study comparisons.

benefits from the *Type-associated Encoder* which adaptively processes the nodes with different types and extracts a better summarization of the source code. The downgraded performance of TAG-MV and TAG-CD indicates the advantages of the type-restricted masking vector and Copying Decay Strategy. These together ensure the accurate execution of the copy and word selection. The comparison of TAG and TAG-RL shows the necessity of the HRL for the training of our framework.

7.2.3 Case Study

In order to show the effectiveness of our framework in a more obvious way, some cases generated by TAG are shown in Table 4. SQL and Python are taken as the targeted programming languages. The comments generated by TAG show great improvements when compared to the baselines. Specifically, for the case in SQL, the keyword “Otkrytie Area” is missing in all the baselines but accurately generated by our framework. For the case in Python, the comment generated by TAG is more readable than the others. These cases demonstrate the high quality of the comments generated by our TAG framework.

8 Conclusion

In this paper, we present a Type Auxiliary Guiding encoder-decoder framework for the code comment generation task. Our proposed framework takes full advantage of the type information associated with the code through the well designed *Type-associated Encoder* and *Type-restricted Decoder*. In addition, a hierarchical reinforcement learning method is provided for the training of our framework. The experimental results demonstrate significant improvements over state-of-the-art approaches and strong applicable potential in software development. Our proposed framework also verifies the necessity of the type information in the code translation related tasks with a practical framework and good results. As future work, we will extend our framework to more complex contexts by devising efficient learning algorithms.

Acknowledgments

This research was supported in part by Natural Science Foundation of China (61876043, 61976052), Natural Science Foundation of Guangdong (2014A030306004, 2014A030308008), Science and Technology Planning Project of Guangzhou (201902010058). Besides, this project is also partly supported by the National Research Foundation, Prime Minister’s Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. This research was also made possible by NPRP grant NPRP10-0208-170408 from the Qatar National Research Fund (a member of Qatar Foundation). The findings herein reflect the work, and are solely the responsibility of the authors.

References

Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. 2002. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*, pages 235–241. IEEE.

Miltiadis Allamanis, Hao Peng, and Charles Sutton.

2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. 2018. An encoder-decoder framework translating natural language to database queries. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 3977–3983. AAAI Press.

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

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833, Berlin, Germany. Association for Computational Linguistics.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824*.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Emil Julius Gumbel. 1954. *Statistical theory of extreme values and some practical applications: a series of lectures*, volume 33. US Government Printing Office.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany. Association for Computational Linguistics.

Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K Reddy. 2019. Deep reinforcement learning for sequence-to-sequence models. *IEEE Transactions on Neural Networks and Learning Systems*.

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

Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. 2017. Adversarial learning for neural dialogue generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2157–2169, Copenhagen, Denmark. Association for Computational Linguistics.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany. Association for Computational Linguistics.

Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 35–40, Sofia, Bulgaria. Association for Computational Linguistics.

Andrew Y Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

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

Ted Tenny. 1988. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407. ACM.

Yuk Wah Wong and Raymond Mooney. 2007. Generation by inverting a semantic parser that uses statistical machine translation. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 172–179, Rochester, New York. Association for Computational Linguistics.

Lijun Wu, Fei Tian, Tao Qin, Jianhuang Lai, and Tie-Yan Liu. 2018a. A study of reinforcement learning for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3612–3621, Brussels, Belgium. Association for Computational Linguistics.

Lijun Wu, Yingce Xia, Fei Tian, Li Zhao, Tao Qin, Jianhuang Lai, and Tie-Yan Liu. 2018b. Adversarial neural machine translation. In *Asian Conference on Machine Learning*, pages 534–549.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. 2018a. SQL-to-text generation with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 931–936, Brussels, Belgium. Association for Computational Linguistics.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018b. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*.

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 *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.