A Grammar-Based Structural CNN Decoder for Code Generation

Zeyu Sun,[†] Qihao Zhu,[†] Lili Mou,[‡] Yingfei Xiong,[*†] Ge Li,[†] Lu Zhang[†]
[†]Key Laboratory of High Confidence Software Technologies (Peking University), MoE;
Software Institute, Peking University, 100871, P. R. China
{szy_, zhuqh, xiongyf, lige, zhanglucs}@pku.edu.cn
[‡]AdeptMind Research, Toronto, ON, Canada
doublepower.mou@gmail.com

Abstract

Code generation maps a program description to executable source code in a programming language. Existing approaches mainly rely on a recurrent neural network (RNN) as the decoder. However, we find that a program contains significantly more tokens than a natural language sentence, and thus it may be inappropriate for RNN to capture such a long sequence. In this paper, we propose a grammar-based structural convolutional neural network (CNN) for code generation. Our model generates a program by predicting the grammar rules of the programming language; we design several CNN modules, including the tree-based convolution and pre-order convolution, whose information is further aggregated by dedicated attentive pooling layers. Experimental results on the HearthStone benchmark dataset show that our CNN code generator significantly outperforms the previous state-of-the-art method by 5 percentage points; additional experiments on several semantic parsing tasks demonstrate the robustness of our model. We also conduct in-depth ablation test to better understand each component of our model.

Introduction

Generating code from natural language description is an important but challenging task in artificial intelligence (Ling et al. 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Mei and Zhang 2018). It is beneficial to various applications. For example, a programmer would like to "open the file, F1" in Python, but does not know how to implement it in the programming language. Hopefully, he or she can obtain the target code "`f = open('F1', 'r')`" by code generation.

With the prosperity of deep learning, the encoder-decoder framework becomes a prevailing approach to sequence generation. In particular, recurrent neural networks (RNNs) typically serve as the encoder and decoder; such architecture is also known as a sequence-to-sequence (Seq2Seq) model (Sutskever, Vinyals, and Le 2014). When applied to code generation, it takes the program description as the input sequence and generates the desired code as the output sequence (Ling et al. 2016).

[*]Yingfei Xiong is the corresponding author. Our code is available at https://github.com/zysszy/GrammarCNN

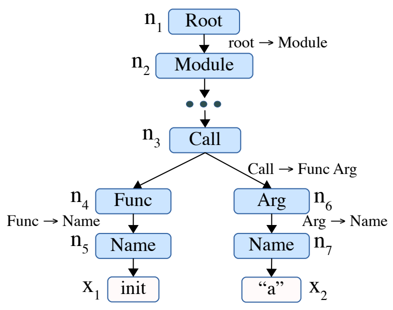

Figure 1: The abstract syntax tree (AST) of code: `init(a)`.

It has been pointed out that programs contain rich structural information, which is important to program modeling (Rabinovich, Stern, and Klein 2017; Yin and Neubig 2017). However, traditional Seq2Seq neural networks do not explicitly model program structures. Figure 1 shows an example of a Python abstract syntax tree (AST), where the two nodes n_3 and n_6 should have interacted intensively as parent-child nodes, but are far away from each other if the tree is pre-order traversed to a sequence. This brings difficulties to Seq2Seq models.

To address this problem, Dong and Lapata (2016) proposed an approach that generates code along the abstract syntax tree (AST) of a program, but their generation is still in the token level. More recent work generates programs by predicting the grammar rule or rewriting rule to apply at each step (Xiong et al. 2018; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017); thus, the generated programs are guaranteed to be syntactically correct. When neural network is used in those approaches, an RNN is used to capture the *autoregressiveness*[1] of predictions within the decoder.

In the deep learning community, researchers are showing growing interest in using the convolutional neural network (CNN) as the decoder (Gehring et al. 2017; Chaturvedi, Pandit, and Garain 2018), because of its efficiency and easiness of training. We further observe that a program is much larger than a natural language sentence and that RNNs—

[1]By "autoregressive," we mean that, during decoding, a step of prediction is dependent on previous decoded steps.

Rule	Explanation
`stmt` $\rightarrow$ `If	For	...`	A statement (`stmt`) could be an `If`-block, a `For`-block, and many others. They are different rules, and the network predicts the most appropriate rule to apply at a time step.
`If` $\rightarrow$ `expr stmt* stmt*`	An `If`-block starts with a testing expression (`expr`). If it is true, the first statement list is executed, or otherwise, the second statement list is executed. In the official python grammar, a rule may generate a list of tokens (e.g., `stmt*`) with an arbitrary length. We examine the training samples and treat each length as a separate rule to predict.

Table 1: Examples of python grammar rules.

even with long short-term memory (Hochreiter and Schmidhuber 1997, LSTM) units—suffer from the long dependency problem (Bengio, Simard, and Frasconi 1994). CNNs, on the contrary, are able to capture features effectively at different regions by sliding windows.

To this end, we propose a grammar-based structural CNN for code generation. Our model generates code by grammar rules of construction in AST, e.g., `If` $\rightarrow$ `expr stmt* stmt*`, following the framework in our previous work (Xiong et al. 2018). Since the sequence of child nodes is generated by one step of prediction, it enables more compact prediction than the token-by-token generation. In other words, our model predicts the sequence of grammar rules, which eventually form an entire program.

In our approach, the prediction of a grammar rule is mainly based on three types of information: the source sequence that specifies the program to be generated, the previously predicted grammar rules, and the partial AST that has been generated. Here, the former one is the input to the encoder. The latter two enable the autoregressiveness of the decoder, and as usual, the decoder is also conditioned on the encoder.

We design several distinct components for the structural CNN, suited to program generation: (1) We first adopt an idea of tree-based convolution that applies sliding windows on the AST structures (Mou et al. 2016). Then we design another CNN module to the pre-order traversal of nodes in the partial AST. These two types of CNNs capture neighboring information not only in the sequence but also in the tree structure. (2) To enhance "autoregressiveness," we apply another CNN module to the ancestors of the node to be generated, and thus the network is aware of where to generate at a certain step. (3) We design a dedicated attentive pooling mechanism that aggregates CNN features interacting with different neural modules. In particular, we find it useful to consider the scope names (e.g., function and method names) during code generation, and use such information as the controller of several attentive pooling layers.

We conducted experiments on an established benchmark dataset, HearthStone, for python code generation (Ling et al. 2016). Experimental results show that our CNN-based code generator outperforms previous RNN approaches to a large extent. We further evaluate our approach on two semantic parsing tasks, where the target programs are shorter than HearthStone; our approach also achieves comparable results to previous state-of-the-art methods, indicating the robustness of our method. We conducted extensive ablation tests, showing that our design of grammar-based structural CNN is better than applying CNN in a naïve fashion.

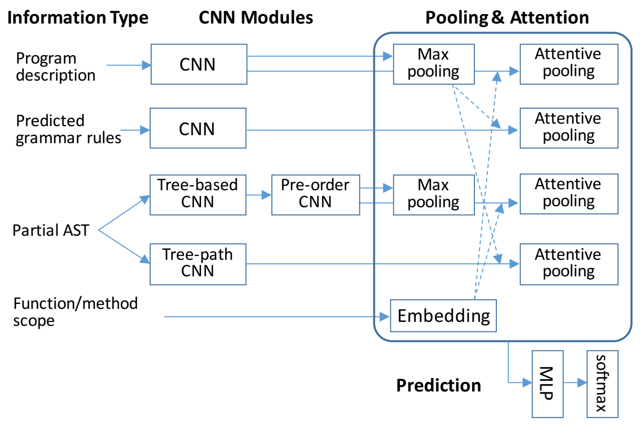

Figure 2: Overview of our model. The dashed arrows indicate attention controllers.

To the best of our knowledge, we are the first to successfully apply CNN decoders for code generation.

The Proposed Model

Figure 2 shows the overall structure of our network. We will first describe the process of grammar-based code generation, and then introduce each module in detail.

Grammar-Based Code Generation

For an input of program description, our task is to generate a piece of executable code that complies with the description. In traditional Seq2Seq models, a program can be represented as a sequence of tokens $\mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_T$, and these tokens are generated in sequence.

Alternatively, a valid program can be represented by an abstract syntax tree (AST) in Figure 1. Leaf nodes are the terminal symbols, denoted as $\mathrm{x}_1, \cdots, \mathrm{x}_T$. Non-leaf nodes are non-terminal symbols $\mathrm{n}_1, \cdots, \mathrm{n}_N$, each representing an abstract component of the program (e.g., an `If`-block). Moreover, child nodes $\mathrm{n}_1, \cdots, \mathrm{n}_k$, stemming from their parent node p, are obtained by applying some grammar rule r, denoted as $\mathrm{p} \xrightarrow{\mathrm{r}} \mathrm{n}_1 \cdots \mathrm{n}_k$. In our work, leaf nodes of different user-defined variables are treated as separate grammar rules by examining the training set. Table 1 illustrates several Python rules and their meanings.[2]

Dong and Lapata (2016) propose to generate an executable command by following the AST, but they predict child nodes $n_1 \cdots n_k$ one at a time with an RNN. In our

[2]Full list available at https://docs.python.org/2/library/ast.html

study, we follow more recent work (Rabinovich, Stern, and Klein 2017; Yin and Neubig 2017; Xiong et al. 2018), predicting the rules $\mathrm{r}_1, \mathrm{r}_2, \cdots, \mathrm{r}_M$ that generate the program. We traverse the tree in depth-first pre-order, and for the first encountered non-terminal symbol, we predict what rule should be used to expand it. In other words, the probability of a program is decomposed as

$$p(\text{program}) = \prod_{n=1}^{M} p(\mathrm{r}_n | \mathrm{r}_1 \cdots, \mathrm{r}_{n-1}) \tag{1}$$

Although a typical programming language contains more grammar rules than distinct AST nodes, grammar-based generation is more compact because the child nodes $\mathrm{c}_1, \cdots, \mathrm{c}_k$ become in place by a single prediction of the rule $\mathrm{p} \xrightarrow{\mathrm{r}} \mathrm{c}_1 \cdots \mathrm{c}_k$. Moreover, the generated program is guaranteed to be syntactically correct.

In the rest of this section, we describe our CNN encoder-decoder model for the prediction of grammar rules.

CNN for the Input

The input of our model is a piece of description that specifies the program to be generated. For code generation of a card in HearthStone, the input is semi-structured data, containing the card's name, properties, and descriptions, illustrated in Figure 4a. For other tasks like semantic parsing (Zettlemoyer and Collins 2005), the input could be a natural language sentence.

We tokenize the input, and obtain a sequence of tokens $\mathrm{x}_1^{(\text{enc})}, \cdots, \mathrm{x}_I^{(\text{enc})}$, where I is the length of the input. The tokens are represented as real-valued vectors $\boldsymbol{x}_1^{(\text{enc})}, \cdots, \boldsymbol{x}_I^{(\text{enc})}$, known as *embeddings*.

Then, a set of convolutional layers are applied and extract features $\boldsymbol{y}_1^{(\text{enc},L)}, \cdots, \boldsymbol{y}_I^{(\text{enc},L)}$. In particular, we adopt shortcut connections every other layer parallel to linear transformation before the activation function, as in ResNet (He et al. 2016). This helps the training of a deep neural network.

Formally, the extracted features $\boldsymbol{y}_i^{(\text{enc},l)}$ are computed by

$$\begin{aligned}\boldsymbol{y}_i^{(\text{enc},l)} = \mathrm{ReLU}\big(& c^{(\text{enc},l)} \cdot \boldsymbol{y}_{i-1}^{(\text{enc},l-2)} \\ &+ W^{(\text{enc},l)}[\boldsymbol{y}_{\lceil i-s \rceil}^{(\text{enc},l-1)}; \cdots; \boldsymbol{y}_{\lceil i+s \rceil}^{(\text{enc},l-1)}]\big)\end{aligned} \tag{2}$$

where $W^{(\text{enc},l)}$ are the convolution weights for the encoder CNN, s is computed by $s = (k-1)/2$, k is the window size (set to 2 in our experiment), and $l = 1, \cdots, L$ indicates the layer in the deep CNN. In particular, $\boldsymbol{y}_i^{(\text{enc},0)}$ is the input embedding $\boldsymbol{x}_i^{(\text{enc})}$. $c^{(\text{enc},l)} = 1$ for even layers and 0 for odd layers, indicating whether the shortcut connection exists for this layer. For the first and last several words, we perform zero padding.

CNN for Predicted Rules

Since the probability of a program is decomposed by grammar rules (Equation 1), we keep track of all previously predicted rules, and build a deep neural network to extract such information.

Let $\mathrm{r}_1, \cdots, \mathrm{r}_{n-1}$ be the previously predicted rules. We embed them as real-valued vectors, $\boldsymbol{r}_1, \cdots, \boldsymbol{r}_{n-1}$, where the embeddings are randomly initialized and learned by backpropagation.

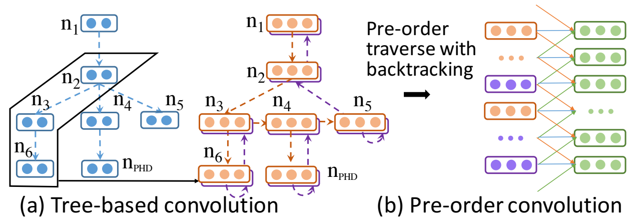
n1
n2
n3
n4
n5
n6
nPHD
Pre-order traverse with backtracking
(a) Tree-based convolution
(b) Pre-order convolution

Figure 3: CNN for the partial AST. The dashed arrows are not a part of the neural network, but indicate topologically neighboring information. In particular, the purple up arrows are backtracking traces.

We apply a deep CNN module with shortcut connections to rule embeddings $\boldsymbol{r}_1, \cdots, \boldsymbol{r}_{n-1}$, extracting features $\boldsymbol{y}_1^{(\text{rule},L)}, \cdots, \boldsymbol{y}_{n-1}^{(\text{rule},L)}$. The computation is the same as Equation 2, but with different weight parameters. Details are not repeated here.

The predicted grammar rules fully specify the generated (partial) program in a compact fashion, which is beneficial for accurate code generation.

However, it is improper to feed the decoder with only predicted rules for autoregressiveness, as they do not provide a concrete/pictorial view of the program due to the compactness. To alleviate this problem, we enhance the decoder with the partial AST, described below.

CNN for Partial AST

An abstract syntax tree (AST) is a tree-structured representation of a program, where a rule $\mathrm{p} \xrightarrow{\mathrm{r}} \mathrm{n}_1 \cdots \mathrm{n}_k$ is expanded as parent-child edges of p and $\mathrm{n}_1, \cdots, \mathrm{n}_k$.

We design a deep CNN module to capture AST's structural information. It contains tree-based convolutional layers, pre-order traversal convolutional layers, as well as a tree-path CNN submodule to inform the network of where the next grammar rule is applied.

Tree-Based CNN. We first apply a tree-based CNN to the partial AST, similar to Mou et al. (2016; 2018). The main intuition is to design a local feature detector of a fixed depth, sliding over a tree to extract structural features, shown in Figure 3a.

The input of tree-based CNN is the partial AST that has been generated, each node represented by an embedding. We also put a placeholder node (n_{PHD} illustrated in Figure 3) to indicate where the next grammar rule is applied.

Suppose a node n_i has a parent node p_i and a grandparent node g_i, and their vector representations are $\boldsymbol{n}_i, \boldsymbol{p}_i$, and $\boldsymbol{g}_i$, respectively. Then the tree-based CNN extracts features $\boldsymbol{y}_1^{(\text{ast})}, \cdots, \boldsymbol{y}_{n-1}^{(\text{ast})}$, computed by

$$\boldsymbol{y}_i^{(\text{ast})} = \mathrm{ReLU}(W^{(\text{ast})}[\boldsymbol{n}_i; \boldsymbol{p}_i; \boldsymbol{g}_i]) \tag{3}$$

where $W^{(\text{ast})}$ is the weight of the tree-based convolution kernel. We pad a special token for the nodes in the top two layers who do not have a parent and/or grandparent.

Note that our tree-based convolution slightly differs from Mou et al. (2016) in that we have a deeper window but do not consider sibling information. This is because our grammar-based generation obtains all siblings at a time by applying a certain rule, and hence, the siblings are less important than ancestors. The complexity of Mou et al. (2016) unfortunately grows exponentially with depth, and is less tractable, whereas our tree-based CNN variant grows linearly. In terms of the convolution computation, we follow Mou et al. (2016) and adopt a perceptron-like interaction. Deep tree-based convolution and shortcut connections as in ResNet could be explored as future work.

Pre-Order Traversal CNN. After obtaining a set of vectors extracted by tree-based CNN, we apply a pre-order traversal convolution with $\boldsymbol{y}^{(\text{ast})}$ being input (Figure 3b). That is, the AST nodes are organized in a sequence by pre-order traversing.

It can be shown that a simple pre-order traverse is not invertible to the tree structure, i.e., different tree structures could yield a same sequence. To address this problem, we keep track of backtracking traces during pre-order traversing. For example, the AST in Figure 3 yields the sequence $\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_3, \mathrm{n}_6, \mathrm{n}_6^{(\text{bt})}, \mathrm{n}_3^{(\text{bt})}, \mathrm{n}_4, \mathrm{n}_4^{(\text{bt})}, \mathrm{n}_{\text{PHD}}, \mathrm{n}_{\text{PHD}}^{(\text{bt})}, \mathrm{n}_5, \mathrm{n}_5^{(\text{bt})}, \mathrm{n}_2^{(\text{bt})}, \mathrm{n}_1^{(\text{bt})}$. Their vector representations (including backtracking nodes and the placeholder node) are predicted by tree-based convolution. Then a deep CNN as in Equation 2 is applied to extract features $\boldsymbol{y}_1^{(\text{tree},L)}, \cdots, \boldsymbol{y}_{2S}^{(\text{tree},L)}$, where L is the number of CNN layers. T is the number of nodes in the AST, and pre-order traverse with backtracking yields $2S$ input units.

It should be noted that the tree-based CNN and the pre-order traversal CNN capture different information. Pre-order traverse yields an order that addresses sequential neighborhood of an AST node during generation, whereas tree-based convolution enables information fusion for nodes that are structurally neighboring. In Figure 3, for example, node n_4 is a child node of n_2. However, after the generation of some other part of the program (namely, n_3 and n_6), the nodes n_2 and n_4 are no longer close to each other. Tree-based convolution directly builds a feature extractor for a node and its ancestors to enable their interaction. Therefore, we believe these two types of CNNs are complementary to each other.

Tree-Path CNN. Should we only consider the above CNNs, it would be hard for the model to tell the position where the next grammar rule is applied. For example, the tree-based CNN and the pre-order traversal CNN would yield very similar features if we expand n_4 or n_5 in Figure 3, despite the placeholder we introduce for pre-order CNN.

Technically speaking, if we follow leftmost derivation, then where the next rule is applied is unambiguous. But such clue is too implicit and should be modeled more explicitly.

We thus extract the path from the root to the node to expand. For example, if we are about to expand n_4, the path should be $\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_4$. Then a set of convolutional layers extract features $\boldsymbol{y}_1^{(\text{path},L)}, \cdots, \boldsymbol{y}_J^{(\text{path},L)}$, also computed as Equation 2. (J is the number of nodes in the path, and L is the number of CNN layers.) We call this *tree-path convolution*.

Pooling and Attention Mechanisms

CNNs extract a set of features with the same size or shape as input. To facilitate softmax prediction for code generation, we need to aggregate information into one or a few fixed-size vectors, regardless of the input size.

Traditionally, people use max pooling for CNNs (Krizhevsky, Sutskever, and Hinton 2012) as well as tree-based CNNs (Mou et al. 2016; Mou and Jin 2018). However, this makes the underlying CNN modules separate and unable to communicate during information aggregation.

Therefore, we incorporate attention mechanisms for CNN pooling, similar to Yu et al. (2018). Essentially, an attention mechanism computes a weighted sum of a set of candidate features (extracted by CNN), where the weights are computed by a controlling vector (for example, a max pooling vector for another CNN module).

Formally, given a controlling vector $\boldsymbol{c}$ and a set of candidate convolutional features $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_D$ extracted by a CNN module (D is the number of feature vectors), we compute attention logit by

$$\widetilde{\alpha}_i = \boldsymbol{y}_i^\top W^{(\text{att})} \boldsymbol{c} \tag{4}$$

where $W^{(\text{att})}$ is a trainable matrix, inspired by metric learning. Then the attention weight α_i for the node i is

$$\alpha_i = \frac{\exp\{\widetilde{\alpha}_i\}}{\sum_{j=1}^{D} \exp\{\widetilde{\alpha}_j\}} \tag{5}$$

Finally, the attentive pooling yields a vector $\boldsymbol{y}^{(\text{att})}$ by

$$\boldsymbol{y}^{(\text{att})} = \sum_{i=1}^{D} \alpha_i \boldsymbol{y}_i \tag{6}$$

To apply such an attentive pooling layer to our underlying CNNs, we consider several key information as the controlling vector. (1) The input description specifies the program to be generated, and we use it to control the grammar rule CNN and the tree-path CNN. In particular, we apply a max pooling layer to aggregate input CNN features as a fixed-size controlling vector, which is used to compute the attention weights for tree-path CNN and the CNN for predicted grammar rules. (2) We note that a *scope name* (namely, a function name or a method name) provides illuminating information about its descendants. Such information is not captured by AST node types, and thus we embed the scope name as a vector and use it to control the pre-order traversal CNN and the CNN for the input. It should be noted that if the current program snippet is under two or more scopes (a function and a method), we only consider the nearest scope as the controlling vector. If the code snippet does not belong to any function or class, then the scope embedding is set to a zero vector.

In addition to the attentive pooling for the tree-based convolution, it is useful to apply another max pooling layer to the pre-order traversal CNN features. Our empirical finding is that the controlling scope embedding makes the attention too peaked at the corresponding AST node, and that aggregated information is not sufficient. Another max pooling layer could preserve more information regardless of the controlling vector.

(a)

```
[NAME]
Acidic Swamp Ooze
[ATK] 3
[DEF] 2
[COST]  2
[DUR]  -1
[TYPE]  Minion
[CLASS] Neutral
[RACE] NIL
[RARITY] Common
[DESCRIPTION]
"Battlecry: Destroy Your Opponent's Weapon"
```

(b)

```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
                CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
                battlecry=Battlecry(Destroy(),
                          WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```

Figure 4: Example card of HearthStone. (a) Input description; (b) Output program.

We would also like to point out that there are different choices of designing the attention mechanism and its controlling connections in a deep neural network with multiple modules. For example, we might as well use the CNN for the input to control all other modules, following the spirit of attention in the encoder-decoder framework. However, our pilot experiment shows that such design yields a worse performance, and thus we adopt the current architecture.

Training and Inference

We concatenate all max pooling and attentive pooling layers. They are fed to a two-layer perceptron, where the last layer has a softmax activation function for predicting the next grammar rule, given by

$$p(\mathrm{r}_i|\cdot) = \frac{\exp\{h_i^{(\text{MLP})}\}}{\sum_{j=1}^{R}\exp\{h_j^{(\text{MLP})}\}} \tag{7}$$

where $h_i^{(\text{MLP})}$ is the input logit of softmax, and R is the number of candidate grammar rules.

Our model is trained by cross-entropy loss against the groundtruth program. Since our entire model is differentiable, all parameters are learned by gradient-based update.

For inference, we seek a sequence of grammar rules that maximizes the probability conditioned on input. The recursive prediction of the rules terminates if every leaf node in the (partial) tree is a terminal symbol. We use beam search to approximate the global inference, and the beam size is 5 in our experiments. Invalid rules for a particular node type are not considered during inference. For example, $\mathrm{p}_2 \to \mathrm{c}_1\mathrm{c}_2$ cannot be applied to the node p_1 if $\mathrm{p}_1 \neq \mathrm{p}_2$.

Evaluation

In this section, we present experimental results of our CNN-based code generation. We evaluated our method on two types of tasks: (1) Python code generation for the HearthStone game, and (2) executable logic form generation for semantic parsing.

	Dataset		
Statistics	**HS**	**ATIS**	**JOBS**
# Train	533	4,434	500
# Dev	66	491	-
# Test	66	448	140
Avg. tokens in description	35.0	10.6	8.7
Max. tokens in description	76.0	48	22
Avg. tokens in code	83.2	33.9	18.1
Max. tokens in code	403	113	50
Avg. nodes in AST	151.0	47.2	40.0
Max. nodes in AST	744	154	138

Table 2: Statistics of the datasets.

Experiment I: HearthStone Code Generation

Dataset. Our first (and main) experiment is based on an established benchmark dataset, HearthStone (Ling et al. 2016, HS). The dataset comprises 665 different cards of the HearthStone game; the input of each data point is a semi-structured description of fields, such as the card name, cost, attack, description, and other attributes; and the output is a Python code snippet that implements the functionality of the card, shown in Figure 4. We follow the train-dev-test split as in Ling et al. (2016). The column HS in Table 2 lists relevant statistics of the dataset.

Metrics. We evaluated our approach by accuracy and BLEU scores. Ideally, the accuracy should count the fraction of functionally correct programs, which unfortunately is not Turing computable. We followed most previous studies (Ling et al. 2016; Yin and Neubig 2017), and calculated the accuracy based on string match (denoted as **StrAcc**).[3] We also find that several generated programs use a different variable name but implements a correct functionality, and that sometimes an argument name in a function call is or is not specified. Although different from the reference program, they are obviously correct programs after manual inspection, and we denote human-adjusted accuracy by **Acc+**. Here, we did not perform checking for non-obvious alternative implementation of an algorithm, and thus Acc+ is still a lower bound of functional accuracy.

The quality of the generated code is further evaluated by the **BLEU** score as an auxiliary metric, which computes how close the generated code is to the groundtruth code in terms of n-grams.

Settings. For the input descriptions, we replace all punctuations with a space; all letters are lower cased. For the neural network, we set the number of CNN layers L to 21, where

[3]Since spaces and empty lines are not represented in AST, the string match is computed based on a "normalized" format. Notice that indentation is crucial to a python program, which has been implicitly captured by AST.

Model	StrAcc	Acc+	BLEU
LPN (Ling et al. 2016)	6.1	–	67.1
SEQ2TREE (Dong and Lapata 2016)	1.5	–	53.4
SNM (Yin and Neubig 2017)	16.2	∼18.2	75.8
ASN (Rabinovich, Stern, and Klein 2017)	18.2	–	77.6
ASN+SUPATT (Rabinovich, Stern, and Klein 2017)	22.7	–	79.2
Our system	**27.3**	**30.3**	**79.6**

Table 3: Performance of our model in comparison with previous state-of-the-art results. Accuracies are in percentage. Yin and Neubig (2017) report an approximately 2% percent boost after human adjustment.

Line #	Model Variant	Acc+	BLEU
1	Full model	**30.3**	79.6
2	Pre-order CNN $\rightarrow$ LSTM	21.2	78.8
3	– Predicted rule CNN	24.2	79.2
4	– Pre-order CNN	25.8	**80.4**
5	– Tree-based CNN	25.8	79.4
6	– Tree-path CNN	28.8	**80.4**
7	– Attentive pooling	24.2	79.3
8	– Scope name	25.8	78.6

Table 4: Ablation test.

the bottom layer does not have skipping connections. We also find it helpful to build a separate network (i.e., the same architecture, but with different weight parameters) for different AST node types (namely, nonterminal nodes, variable nodes, and function name nodes). This enables us to be better aware of node types during generation. When predicting variable nodes, we introduce a new softmax target, for each slot, that could copy the slot value. The layers of difference CNN modules are set to the same dimension, chosen by validation from {128, 192, 256} for each predictor network. We applied dropout (drop rate$= 0.5$) and ℓ_2 penalty to regularize the fully connected layers. The network is trained by the Adam optimizer (Kingma and Ba 2015) with default hyperparameters.

Overall Results. Table 3 presents the results of our CNN-based code generation, in comparison with previous state-of-the-art models: (1) Latent Predictor Network (Ling et al. 2016, LPN), an enhanced sequence-to-sequence model with multiple token-level predictors; (2) SEQ2TREE (Dong and Lapata 2016), a sequence-to-sequence model based on AST; (3) Syntactic Neural Model (Yin and Neubig 2017, SNM), an LSTM decoder based on AST structures; and (4) Abstract Syntax Networks (Rabinovich, Stern, and Klein 2017, ASN), another AST-based sequence-to-sequence model, which builds two LSTMs predicting rules in the horizontal and vertical directions, respectively. The ASN model has a variant (ASN+SUPATT) where attention is trained in a supervised fashion.

As shown, our model outperforms all previous results in terms of both accuracy and BLEU scores. In particular, our accuracy is significantly higher than the previous state-of-the-art result by about 5 percentage points in terms of string accuracy. For human adjusted accuracy (Acc+), Yin and Neubig (2017) report approximately 2 percentage points improvement. Similar phenomena are observed in our scenario, and we achieve an Acc+ score of 30.3%, showing strong evidence of the effectiveness of our approach.

```
Generated Code:
class Gnoll(MinionCard):
    def __init__(self):
        super().__init__("Gnoll", 2, CHARACTER_CLASS.ALL,
                 CARD_RARITY.COMMON, False)

    def create_minion(self, p):
        return Minion(2, 2, taunt = True)

Reference Code:
class Gnoll(MinionCard):
    def __init__(self):
        super().__init__("Gnoll", 2, CHARACTER_CLASS.ALL,
                 CARD_RARITY.COMMON, False)

    def create_minion(self, player):
        return Minion(2, 2, taunt = True)

Reference Code For Anthor Card:
class DefenderMinion(MinionCard):
    def __init__(self):
        super().__init__("Defender", 1, CHARACTER_CLASS.PALADIN,
                 CARD_RARITY.COMMON)

    def create_minion(self, p):
        return Minion(2, 1)
```

Figure 5: Example of the generated code. (Compared with reference codes, the code we generated has a different variable, but a correct functionality.)

We find an intriguing fact that several previous methods could achieve a similar BLEU score to our approach, but with a much lower accuracy. For example, the ASN model has a BLEU score of 79.2, comparable to 79.6 given by our model. However, ASN only achieves 22.7% string accuracy, whereas ours is 27.3%. This is because the BLEU metric only measures surface n-gram similarity of programs. Previous methods (like ASN) are able to generate seemingly plausible code, which are in fact incorrect in terms of details. Therefore, we only consider BLEU scores (adopted in previous work) as a secondary metric. The major metric, namely, accuracy, shows that our approach generates much more accurate programs than previous models.

Ablation test. We conducted extensive ablation tests to analyze the contribution of each component. Although the development of our network started from a simple baseline and we incrementally added on useful components, the ablation test was conducted in an opposite way: it started from the full model, and we either removed a single component of our model or substituted it with some reasonable alternatives. We report results of our ablation test in Table 4.

We first analyze the effect of CNN by substituting a CNN component with LSTM-based RNN (Lines 1 & 2). Since the main information lies in the partial AST (151 nodes on average shown in Table 2) as opposed to, say, the predicted grammar rules, we replace only the pre-order traversal CNN to an LSTM in this controlled experiment. Such setting achieves 1 point lower BLEU, but 9 percent lower accuracy. The result is consistent with previous models in the literature (Table 3),

Input description: list airport in ci0
Output λ-calculus:

```
lambda $0 e ( and ( airport $0 )
                  ( loc:t $0 ci0 ) )
```

Figure 6: Example of the ATIS dataset for semantic parsing.

where RNN is the main building block achieving lower accuracy.

The scenario can be better understood if we consider the setting where we simply remove the pre-order traversal CNN (Line 4, Table 4). Both based on a tree-based CNN layer, the LSTM component yields a 4% lower accuracy than simply without LSTM. This implies that RNNs are not suitable to this task, which is probably because a program contains too many tokens and AST nodes. An RNN applied to such a long sequence can be very difficult to train (Pascanu, Mikolov, and Bengio 2013), achieving significantly worse performance.

We also analyze other components of our model, including the CNN for predicted rules, the tree-based convolutional layer, the tree-path convolution, the attentive pooling mechanism, and the scope controllers for pooling (Lines 3–8, Table 4). We see that each of the above components contributes to the whole model in its own way, improving the accuracy by 3–6 percent. These results show that we have designed reasonable components of the neural architecture, suited to the code generation task.

Experiment II: Semantic Parsing

Dataset and Settings. Semantic parsing aims to generate logical forms given a natural language description. It can be thought of as code generation for a domain-specific language, because the logical form is an executable, unambiguous formal language. However, the style of semantic parsing differs significantly from python code generation. Since our model is mainly developed on the HS dataset, this experiment serves as additional evaluation of the generalizability of our model.

We evaluated our model on two semantic parsing datasets (ATIS and JOBS) used in Dong and Lapata (2016), where the input is a natural language sentence. The output of ATIS is in the λ-calculus form (illustrated in Figure 6), while for JOBS, it is in the Prolog-style form. We used the standard train-dev-test split for the datasets Zettlemoyer and Collins (2005). We see from the statistics in Table 2 that the logical forms for semantic parsing contain significantly fewer nodes and tokens than HS Python code.

We adopted mostly the same network from the HearthStone experiment to semantic parsing. The number of layers L is 7 in this experiment. We did not build a separate network for different node types as our network is prone to overfitting for such small datasets. Besides, we introduced the pointer network (See, Liu, and Manning 2017) to copy variable names (e.g., `ci0` in Figure 6) due to the property of the datasets, which is also the practice of previous work (Rabinovich, Stern, and Klein 2017).

	ATIS		JOBS	
	System	**Accuracy**	**System**	**Accuracy**
Traditional	ZH15	84.2	ZH15	85.0
	ZC07	84.6	PEK03	88.0
	WKZ14	**91.3**	LJK13	90.7
Neural	SEQ2TREE	84.6	SEQ2TREE	90.0
	ASN	85.3	ASN	91.4
	ASN-SUPATT	85.9	ASN-SUPATT	**92.9**
	Our System	85.0	Our System	89.3

Table 5: Accuracy in semantic parsing (in percentage).

Results. We followed Dong and Lapata (2016) and evaluated our approaches by accuracy. It counts the fraction of exact match, except that we adjusted the order of conjunction and disjunction clauses to avoid spurious errors, as in all previous work. We did not measure BLEU since it is not used in existing studies.

Table 5 shows the performance of our model. As seen, neural models are generally worse than the WKZ14 system (Wang, Kwiatkowski, and Zettlemoyer 2014), which uses a large number of rules and templates, but they outperform other traditional semantic parsing systems, including ZH15 (Zhao and Huang 2015), ZC07 (Zettlemoyer and Collins 2007), PEK03 (Popescu, Etzioni, and Kautz 2003), and LJK13 (Liang, Jordan, and Klein 2013).

We also see that our grammar-based structural CNN decoder achieves similar results to the state-of-the-art neural models (Dong and Lapata 2016; Rabinovich, Stern, and Klein 2017). It should also be pointed out that in semantic parsing, we do not achieve a large performance boost as in HearthStone (HS) code generation. This is probably because the logic form for semantic parsing is usually short, containing only 1/4–1/3 tokens as in HS, and thus, both RNN and CNN are fine for logic form generation. This experiment nevertheless provides additional evidence of the generalizability and flexibility of our CNN code generation, since our model is basically designed for long programs (such as HS) but also works fine with semantic parsing.

Related Work

Early studies on code generation mostly focus on domain specific languages (Zettlemoyer and Collins 2005; Kushman and Barzilay 2013; Wang, Kwiatkowski, and Zettlemoyer 2014). They are largely based on rules and human defined features, and thus are highly restricted.

Recently, researchers introduce neural networks to generate code in a general-purpose programming language. Ling et al. (2016) adopt a sequence-to-sequence model, but enhance it with multiple predictors. Other studies generate programs along abstract syntax trees (Dong and Lapata 2016; Rabinovich, Stern, and Klein 2017; Yin and Neubig 2017). However, their decoders are all based on RNNs, which are shown improper for code generation in our experiments.

CNNs are origianlly used in classification tasks (Lecun and Bengio 1995; Krizhevsky, Sutskever, and Hinton 2012). Mou et al. (2016) propose a tree-based CNN to capture structural information. Such idea can be extended to general graphs, e.g., molecule analysis (Duvenaud et al. 2015). Recently, researchers develop deep CNNs for decoders (Gehring et al. 2017; Chaturvedi, Pandit, and Garain 2018). In our paper, we incorporate the idea of structure-sensitive CNN and CNN for generation, and design a grammar-based structural CNN for code generation.

Conclusion

In this paper, we propose a grammar-based structural CNN for code generation. Our model makes use of the abstract syntax tree (AST) of a program, and generates code by predicting the grammar rules. We address the problem that traditional RNN-based approaches may not be suitable to program generation, possibly due to the large number of tokens/nodes in a program. We thus design a CNN encoder-decoder model based on AST structures.

Our main experiment on the HearthStone dataset shows that we have achieved significantly better performance than previous RNN-based methods. Additional experiments on two semantic parsing tasks demonstrate the robustness of our approach. We also conducted in-depth ablation test to verify the effectiveness of each component in our model.

Acknowledgments

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, and National Natural Science Foundation of China under Grant Nos. 61672045 and 61529201.

References

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* 5(2):157–166.

Chaturvedi, A.; Pandit, O.; and Garain, U. 2018. CNN for text-based multiple choice question answering. In *ACL*.

Dong, L., and Lapata, M. 2016. Language to logical form with neural attention. In *ACL*, 33–43.

Duvenaud, D. K.; Maclaurin, D.; Iparraguirre, J.; Bombarell, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2224–2232.

Gehring, J.; Auli, M.; Grangier, D.; Yarats, D.; and Dauphin, Y. N. 2017. Convolutional sequence to sequence learning. In *ICML*, 1243–1252.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *CVPR*, 770–778.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Kingma, D. P., and Ba, L. 2015. J. ADAM: a method for stochastic optimization. In *ICLR*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*, 1097–1105.

Kushman, N., and Barzilay, R. 2013. Using semantic unification to generate regular expressions from natural language. In *NAACL*, 826–836.

Lecun, Y., and Bengio, Y. 1995. Convolutional networks for images, speech, and time-series. In *The Handbook of Brain Theory and Neural Network*. MIT Press.

Liang, P.; Jordan, M. I.; and Klein, D. 2013. Learning dependency-based compositional semantics. *Computational Linguistics* 39(2):389–446.

Ling, W.; Blunsom, P.; Grefenstette, E.; Hermann, K. M.; Kočiskỳ, T.; Wang, F.; and Senior, A. 2016. Latent predictor networks for code generation. In *ACL*, 599–609.

Mei, H., and Zhang, L. 2018. Can big data bring a breakthrough for software automation? *SCIENCE CHINA Information Sciences* 61(5):056101:1–056101:3.

Mou, L., and Jin, Z. 2018. *Tree-Based Convolutional Neural Networks: Principles and Applications*. Springer.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 1287–1293.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. In *ICML*.

Popescu, A.-M.; Etzioni, O.; and Kautz, H. 2003. Towards a theory of natural language interfaces to databases. In *Prof. Int. Conf. Intelligent User Inferfaces*, 149–157.

Rabinovich, M.; Stern, M.; and Klein, D. 2017. Abstract syntax networks for code generation and semantic parsing. In *ACL*, 1139–1149.

See, A.; Liu, P. J.; and Manning, C. D. 2017. Get to the point: Summarization with pointer-generator networks. In *ACL*, 1073–1083.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*.

Wang, A.; Kwiatkowski, T.; and Zettlemoyer, L. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *EMNLP*, 1284–1295.

Xiong, Y.; Wang, B.; Fu, G.; and Zang, L. 2018. Learning to synthesize. In *Proc. 4th Int. Genetic Improvement Workshop, GI@ICSE 2018*, 37–44.

Yin, P., and Neubig, G. 2017. A syntactic neural model for general-purpose code generation. In *ACL*, 440–450.

Yu, A. W.; Dohan, D.; Luong, M.-T.; Zhao, R.; Chen, K.; Norouzi, M.; and Le, Q. V. 2018. QANet: Combining local convolution with global self-attention for reading comprehension. In *ICLR*.

Zettlemoyer, L. S., and Collins, M. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *UAI*, 658–666.

Zettlemoyer, L., and Collins, M. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP*.

Zhao, K., and Huang, L. 2015. Type-driven incremental semantic parsing with polymorphism. In *NAACL*, 1416–1421.