Abstract—Generative Adversarial Networks (GANs) have shown great capacity on image generation, in which a discriminative model guides the training of a generative model to construct images that resemble real images. Recently, GANs have been extended from generating images to generating sequences (e.g., poems, music, and codes). Existing GANs on sequence generation mainly focus on general sequences, which are grammar-free. In many real-world applications, however, we need to generate sequences in a formal language with the constraint of its corresponding grammar. For example, to test the performance of a database, one may want to generate a collection of SQL queries, which are not only similar to the queries of real users, but also follow the SQL syntax of the target database. Generating such sequences is highly challenging because both the generator and discriminator of GANs need to consider the structure of the sequences and the given grammar in the formal language.

To address these issues, we study the problem of syntax-aware sequence generation with GANs, in which a collection of real sequences and a set of pre-defined grammatical rules are given to both discriminator and generator. We propose a novel GAN framework, namely TreeGAN, to incorporate a given Context-Free Grammar (CFG) into the sequence generation process. In TreeGAN, the generator employs a recurrent neural network (RNN) to construct a parse tree. Each generated parse tree can then be translated to a valid sequence of the given grammar. The discriminator uses a tree-structured RNN to distinguish the generated trees from real trees. We show that TreeGAN can generate sequences for any CFG and its generation fully conforms with the given syntax. Experiments on synthetic and real data sets demonstrated that TreeGAN significantly improves the quality of the sequence generation in context-free languages.

Index Terms—Generative Adversarial Networks, GANs, Tree Generation, Sequence Generation, Context-Free Language
Chomsky hierarchy [7], which can apply to many existing formal languages. A formal definition of CFGs are provided in Section II-B. To the best of our knowledge, we make the very first effort to build a syntax aware GAN for sequence generation.

Although GANs have been successfully applied on many tasks, learning such a syntax-aware generative network is not an easy task, which has several challenges:

- **Guarantee the syntax correctness**: The difficulty of ensuring the syntax validity lies in the nature of sequence generator: it generates tokens one by one in a sequential order. Most syntax models employ a top-down structure like trees to abstract the grammatical information. To fully achieve the syntax awareness, the sequence generator have to follow a certain grammatical tree structure. However, the structure of grammatical trees can vary a lot, it is impossible for a sequence generator to cover all the possibilities.

- **Tracking the syntax state of incomplete phrase**: RNN is usually used as the generator in sequence generation, which stores a summary of the generated tokens in its hidden state at each step. However, such summary does not keep track of the syntax information in the partially generated sequence, which leads to possible syntax errors in the entire sequence.

- **Syntax-aware discriminator**: Discriminator is a crucial component of a GAN framework, and should be designed specifically based on the nature of studied task.

To tackle above challenges, we propose a novel GAN model called TreeGAN. Instead of generating sequences directly, TreeGAN absorbs a set of grammatical rules and learns to generate parse trees. Each generated tree corresponds to a sequence that is valid according to the given grammar. This approach imposes hard restrictions on the generator, and the syntax correctness of generated sequences is guaranteed.

We show how these restrictions can be applied in Section III-B. Consequently, the vanilla RNN/LSTM is no longer the optimal choice for the discriminator since the generator of TreeGAN is generating trees instead of plain sequences. To better distinguish the fake parse trees from real parse trees, we use TreeLSTM [8] to guide the tree generator during the adversarial training, the details are presented in Sec. III-C. The corresponding pre-training strategies are discussed in Sec. III-D.

II. PROBLEM FORMULATION

A. Syntax Aware Sequence Generation

The syntax-aware sequence generation problem is defined as follows.

**Definition II.1.** Given a dataset of real-world structured sequences $\mathcal{X} = \{X_1, \ldots, X_N\}$, where all $X_n \in \mathcal{X}$ follows a grammar $G$, train a $\theta$-parameterized generative net $G_\theta$ to construct a sequence $Y_{1:T} = (y_1, \ldots, y_T)$ with $y_t \in \mathcal{Y}$, where $\mathcal{Y}$ is the set of vocabulary of tokens.

B. Grammar

In this paper, we study the sequence generation problem in context-free grammars (CFGs), which is formulated in the well-known Chomsky hierarchy [7]. CFGs can apply to many existing formal languages, such as palindrome and SQL. A CFG is formally defined as $G = (V, T, \mathcal{P}, S)$, where $V$ is a set of non-terminal variables, $T = \mathcal{Y} \cup \{\epsilon\}$ the set of terminal variables, $\mathcal{P}$ the set of production rules, and $S \in V$ the start symbol. Each production rule $P \in \mathcal{P}$ follows the form:

$$V \mapsto (T \cup \mathcal{V})^+$$

C. Parse Tree

For each derivation of a CFG sequence, there is a corresponding tree representation called parse tree. The parse tree for any sequence follow context free grammar $G = (V, T, \mathcal{P}, S)$ are trees with following properties:

1. The root node is labeled by $S$.
2. The interior node is labeled by a variable in $V$.
3. Every leaf is labeled by a terminal in $T$.
4. If a node labeled $A$, and its children are labeled $N_1, \ldots, N_k$ from left to right, then $A \Rightarrow N_1, \ldots, N_k$ is a production rule in $\mathcal{P}$.

If we concatenate leaves of a parse tree from left to right and top to bottom, we obtain a yield of the tree, which is equivalent to the string derived from the root variable.

**Theorem 1.** Let $G = (V, T, \mathcal{P}, S)$ be a CFG. If a sequence $Y$ can be derived using the production rules from $\mathcal{P}$ and the derivation starts with $S$, then there is a parse tree with root $S$ that yields $Y$.

**Proof.** It is equivalent to the proof of Theorem 5.12 in [9].

**Lemma 1.** If sequence $X$ follows a context free grammar $G = (V, T, \mathcal{P}, S)$, there is a sequence of productions $Z = (P_1, \ldots, P_k)$ that derives $Y$, where $P_1, \ldots, P_k \in \mathcal{P}$. Such mapping can be denoted as $Z \Rightarrow X$.

Given Lemma 1, we can find a set of production sequences $\mathcal{D} = \{D_1, \ldots, D_N\}$ for $\mathcal{X} = \{X_1, \ldots, X_N\}$, where $D_1 \Rightarrow X_1, \ldots, D_N \Rightarrow X_N$. How to parse each $X_n$ into $D_n$ is out of the scope of this paper and will not be discussed here.

Now we can transform the original syntax-aware sequence generation problem defined in Section II-A into a parse tree generation problem.

**Definition II.2 (Parse Tree Generation Problem).** Given a CFG defined as $G = (V, T, \mathcal{P}, S)$, and $\mathcal{D} = \{D_1, \ldots, D_N\}$ where all the production rules in $\{Z_1, \ldots, Z_N\}$ are from $\mathcal{P}$, the goal is to train a $\theta$-parameterized generative net $G_\theta$ to construct a sequence $Z_{1:T} = (P_1, \ldots, P_T)$ with $P_t \in \mathcal{P}$.

Additionally, we also train a $\phi$-parameterized discriminative net $D_\phi$ to guide $G_\theta$ to improve the generating quality. Specifically, $D_\phi(Z)$ is a probability indicating how likely $Z$ is a real data sample.

$^1\epsilon$ denotes the empty token, alternatively it can be considered as a special symbol that not included in the set of terminal variables.
III. METHODOLOGY

A. Generative Adversarial Network

GAN [1] aims to obtain the equilibrium of the following optimization objective

\[ L(\theta, \phi) = -\mathbb{E}_{X \sim p_d} \log D_\theta(X) \\
- \mathbb{E}_{Y \sim G_\phi} \log (1 - D_\phi(Y)) \] (2)

where \( L \) is minimized w.r.t. \( D_\theta \) and is maximized w.r.t. \( G_\phi \). \( X \) are sampled from the real-data distribution \( p_d \). Since the first term of Eq. (2) does not depend on \( G_\phi \), we only need to consider the second term when training the generator. However, applying GAN on sequence data has a problem: the gradient of loss from \( D_\theta \) w.r.t the output of \( G_\phi \) is not meaningful for discrete tokens [1, 2]. Thus, we follow the approach proposed in SeqGAN [2] to use the policy gradient [10] to guide the learning of \( G_\phi \). The reward of \( G_\phi \) when given a start state \( s_0 \) is:

\[ J(\theta) = \mathbb{E}_{Y \sim G_\phi} \log \left( \frac{G_\phi(y_1|s_0) \prod_{t=2}^{T} G_\phi(y_t|y_{1:t-1})}{R(Y_{1:T})} \right) \] (3)

where \( R(\cdot) \) is the reward function for a generated sequence, here we consider the estimated probability of being real by the discriminative net \( D_\theta \) as the reward. Formally it is defined as

\[ R(Y_{1:T}) = D_\theta(Y_{1:T}) \] (4)

Hence, for sequence generation task, the objective of training the discriminative net is \( \arg \min_\phi L(\theta, \phi) \), where \( \theta \) is fixed. And the objective of training the generative net is arg \( \min_\theta J(\theta) \).

B. Tree Generator

Inspired by the model proposed in [5], we consider the tree generation problem as generating a sequence of actions. The actions can be categorized into two types, which are (1) the production rules as defined in Eq. 1 and (2) the terminal tokens in \( V \). The generation proceeds in depth-first, left-to-right order.

\( G_\theta \) starts from the root node at step \( t_1 \) and proceeds by choosing different production rules to expand the tree, and at leaves, the model generates terminal tokens to close the tree branches.

We employ a vanilla LSTM to implement our tree generator:

\[
\begin{align*}
    i_t &= \sigma(W^{(i)}_t x_t + U^{(i)}_t h_{t-1} + b^{(i)}), \\
    f_t &= \sigma(W^{(f)}_t x_t + U^{(f)}_t h_{t-1} + b^{(f)}), \\
    o_t &= \sigma(W^{(o)}_t x_t + U^{(o)}_t h_{t-1} + b^{(o)}), \\
    u_t &= \tanh(W^{(u)}_t x_t + U^{(u)}_t h_{t-1} + b^{(u)}), \\
    c_t &= i_t \odot u_t + f_t \odot c_{t-1}, \\
    h_t &= o_t \odot \tanh(c_t),
\end{align*}
\] (5)

where, \( i_t, f_t, o_t, c_t, u_t \) are the input gate, the forget get, the output gate, the memory cell and the hidden state at time step \( t \) respectively. \( u_t \) is the memory cell before input gate at step \( t \), and \( \odot \) denotes the element-wise multiplication.

For a data sample \( D = (x_1, \ldots, x_t) \), the input vector at time step \( t \) is \( x_t = (a_{t-1}, p_t) \), where \( a_{t-1} \) is the action embedding vector for \( d_{t-1} \) and \( p_t \) is the parent embedding vector for \( d_t \).

**Action Embedding:** Two action embedding matrices \( W^{(P)} \) and \( W^{(V)} \) are initialized before train the generator \( G_\theta \). Each row in \( W^{(P)} (W^{(V)}) \) corresponds to an embedding vector for an action of production rules (terminal tokens).

**Parent Embedding:** The tree generator uses the parent feeding to inherit the information encoded in the parent action along the generation tree. The parent action step \( p(t) \) is formally defined as the time step at which the action node at step \( t \) is initiated.

**Generation State Tracking:** As we discussed in Section I, the conventional RNN stores lossy summarization in its hidden state \( h_t \), which only contains incomplete syntax information of the generated part of a sequence. Thus, we need an extra control on the RNN to track the generation state accurately. The output of the LSTM at time step \( t \) is denoted as \( o_t \in \mathbb{R}^L \), where \( L \) is the size of the set of actions. At the output layer of each time step, the generator samples an action from the a multinomial distribution denoted by \( \text{softmax}(o_t) = (\hat{o}_t^{(1)}, \ldots, \hat{o}_t^{(L)}) \), where \( \hat{o}_t^{(k)} \) corresponds to the probability of sampling action \( a_k \) at time step \( t \). A mask matrix \( M^{(G)} \in \{0,1\}^{(|V|\times(|P|+|T|))} \) can be derived for grammar \( G \). The \( k \)-th row in \( M^{(G)} \), which is denoted as \( M^{(G)}(k) \), marks the valid actions for \( v_k \in V \) as 1s and the invalid ones as 0s. Thus, when the generator \( G_\theta \) reaches the time step \( t_k \) where the corresponding node is non-terminal node \( v_k \in V \), then the following masking is performed before it generates the token for step \( t_k \):

\[ \tilde{o}_t = \text{softmax}(o_t) \odot M^{(G)}(k) \] (6)

Hence, the probability of invalid actions for \( v_k \) is reset to 0 in \( o_t \). In the other cases, when \( G_\theta \) reaches a time step where the corresponding node is a terminal node \( y_k \in T \), then \( y_k \) is directly generated. By applying such masking process, our tree generator can no longer sample actions that violate the syntax.

**Tracking Algorithm:** The remaining problem is how the tree generator identifies the node type and retrieves the parent action at time step \( t_k \). At the beginning of generation, the stacks are initialized as \( \Omega^{(P)} = [\Gamma, R] \) and \( \Omega^{(C)} = [\Gamma, S] \), where \( \Gamma \) is the empty stack symbol that cannot be popped and \( R \) is the pseudo-root symbol. At each step \( t \) of generation, the following stack operations are performed sequentially:

\[ P \xleftarrow{\text{pop}} \Omega^{(P)}, \quad C \xleftarrow{\text{pop}} \Omega^{(C)} \]

where \( P \) is the corresponding parent action and \( C \) is the head variable for time step \( t_k \). If \( C \in T \), then \( C \) is generated directly and no further stack operations are required before next time step.

When \( C \in V \), the embedding of the action at previous time step \( t_k \) and the embedding of \( P \) are fetched respectively to build the input vector \( x_t = (a_{t-1}, p_t) \). After applying Eq. (5), an action that takes the form \( (C \rightarrow H) \in P \) is generated based upon the masked probability vector \( \tilde{o}_t \), where \( H \in (V \cup T)^+ \) is a sequence of variables. Before moving forward to next time
step, the following stack operations are performed, \( C \xrightarrow{\text{push}} \Omega^{(P)} \), reversed(\( H \xrightarrow{\text{push}} \Omega^{(C)} \)), where we push the variable \( C \) into the parent stack, and push the variables in \( H \) into the children stack in a reversed order.

**Close A Generation:** If \( \Omega^{(P)} = \Omega^{(C)} = \Gamma \) at the beginning of a time step, it indicates that all interior nodes have been expanded and all leaves are labeled with a terminal token in the tree, then the generator closes the generation by producing an end symbol.

### C. Tree Discriminator

Since we require the discriminator encode the rich grammar information of a sequence, it should capture the structure and the semantics of the corresponding parse tree. Thus, we use the Child-Sum Tree-LSTM \( [8] \) as the discriminator of TreeGAN. The formulation is as follows,

\[
\begin{align*}
\hat{h}_j &= \sum_{k \in \text{Ch}(j)} h_k, \\
i_j &= \sigma(W^{(i)}x_j + U^{(i)}\hat{h}_j + b^{(i)}), \\
f_{jk} &= \sigma(W^{(f)}x_j + U^{(f)}h_k + b^{(f)}), \\
o_j &= \sigma(W^{(o)}x_j + U^{(o)}\hat{h}_j + b^{(o)}), \\
u_j &= \tanh(W^{(u)}x_j + U^{(u)}\hat{h}_j + b^{(u)}), \\
c_j &= i_j \odot u_j + \sum_{k \in \text{Ch}(j)} f_{jk} \odot c_k, \\
h_j &= o_j \odot \tanh(c_j),
\end{align*}
\]

where \( \text{Ch}(j) \) refers to the set of children of node \( j \). This model is also called Child-Sum Tree-LSTM, in which a tree proceeds from leaves to the root. Moreover, \( h_r \) denotes the final hidden state for a given tree where \( r \) is the root node of the tree, and it encodes the entire tree and can be used for classification. A fully connected linear layer is appended after the output of Tree-LSTM to obtain the confidence:

\[
\Psi = \text{sigmoid}(W^{(c)}h_r + b^{(c)}),
\]

where \( \Psi \in (0, 1) \) refers to the probability of the encoded tree being a real instance.

### D. Pre-Training

Before starting the adversarial training, pre-training of \( D_o \) and \( G_o \) are usually required to reach a good initialization, which can facilitate the convergence later in adversarial training. We initialize the tree generator parameters using conventional maximum likelihood estimation (MLE). As to the tree discriminator initialization, we let the discriminator distinguish the twisted trees from the real trees. We randomly swap two subtrees of different head types for each real parse tree in the corpus to construct the twisted tree counterparts. The swapping operation breaks the syntax of the real parse tree, which guides the discriminator to learn correct syntax patterns.

### IV. Synthetic Study

#### A. Dataset

We prepare three different synthetic datasets with controlled syntax and schema (for SQL datasets only).

- **PLD**: A dataset of palindrome in English alphabet (26 capital letters and 26 lowercase letters).
- **SQL-A**: A dataset of SQL queries (SELECT queries) with a small set of grammatical rules.
- **SQL-B**: A dataset of SQL queries with a larger set of grammatical rules.

Note that the proposed TreeGAN uses only the grammar (syntax) but not the schema to train the sequence generator.

#### B. Compared Methods

We test the following methods to demonstrate the effectiveness of the proposed method.

- **TreeGAN** (Our): it uses the tree generator described in Sec. III and the Child-Sum Tree-LSTM as the discriminative model.
- **TreeGAN**- (Our): A variation of TreeGAN that uses LSTM as the discriminative model instead of Tree-LSTM.
- **TreeGen** [5]: Tree generator without adversarial training, using MLE training.
- **SeqGAN** [2]: The original Sequence GAN that proposed for general purpose sequence generation task.
- **SeqGAN**- [2]: A variation of Sequence GAN that uses LSTM as the discriminative model instead of CNN.
- **LSTM** [11]: LSTM generator employs Maximum Likelihood Estimation as the training strategy.

All compared methods are implemented using PyTorch\(^2\) in Python. The batch size is set to 64 for all models.

#### C. Experimental Settings

For each dataset we used in this section, we first transform each sequence into a sequence of actions that pre-defined in the given syntax. Then we randomly select 10% of data samples to form the test (reference) set, and use the remaining

\(^2\)http://pytorch.org

---

### TABLE I: Quantitative Results on PLD

| Methods | BLEU | METEOR | ROUGE-L | SYNTAX | SCHEMA |
|---------|------|--------|---------|--------|--------|
| LSTM    | 35.79 | 25.98  | 34.36   | 1.56   |
| SeqGAN- | 40.41 | 28.14  | 34.74   | 2.13   |
| SeqGAN  | 44.57 | 29.04  | 34.81   | 2.57   |
| TreeGen | 42.49 | 27.19  | 34.59   | 100.00 |
| TreeGAN | 56.71 | 27.84  | 40.25   | 100.00 |
| TreeGAN | 59.33 | 30.99  | 42.20   | 100.00 |

### TABLE II: Quantitative Results on SQL-A

| Methods | BLEU | METEOR | ROUGE-L | SYNTAX | SCHEMA |
|---------|------|--------|---------|--------|--------|
| LSTM    | 79.24 | 65.44  | 87.91   | 41.90  | 4.78   |
| SeqGAN- | 80.24 | 75.78  | 91.00   | 45.79  | 6.12   |
| SeqGAN  | 84.10 | 77.01  | 91.42   | 45.71  | 6.88   |
| TreeGen | 86.12 | 72.00  | 90.20   | 100.00 | 12.00  |
| TreeGAN | 87.01 | 79.42  | 93.25   | 100.00 | 13.57  |
| TreeGAN | 88.15 | 80.31  | 96.13   | 100.00 | 15.66  |
TABLE III: Quantitative Results on SQL-B

| Methods   | BLEU  | METEOR | ROUGE-L | SYNTAX | SCHEMA |
|-----------|-------|--------|---------|--------|--------|
| LSTM      | 56.46 | 33.60  | 76.64   | 45.42  | 0.00   |
| SeqGAN-   | 61.54 | 35.79  | 78.23   | 66.42  | 1.24   |
| SeqGAN    | 64.33 | 36.23  | 84.02   | 65.11  | 1.11   |
| TreeGen   | 83.00 | 38.59  | 72.28   | 100.00 | 11.40  |
| TreeGAN   | 86.58 | 39.42  | 80.54   | 100.00 | 17.60  |
| TreeGAN   | 87.41 | 41.02  | 80.46   | 100.00 | 19.25  |

TABLE IV: Quantitative Results on Django

| Methods   | BLEU  | METEOR | ROUGE-L |
|-----------|-------|--------|---------|
| LSTM      | 19.08 | 26.77  | 55.58   |
| SeqGAN-   | 23.74 | 35.56  | 63.82   |
| SeqGAN    | 26.78 | 39.21  | 65.09   |
| TreeGen   | 68.12 | 39.81  | 72.73   |
| TreeGAN   | 71.43 | 40.70  | 75.42   |
| TreeGAN   | 75.10 | 41.55  | 80.48   |

90% as the training set. For all GAN models include the proposed TreeGAN, we perform 50 epochs of pre-training before starting the adversarial training. And the adversarial training last up to 50 epochs or until the policy gradient loss converges. We report the evaluation scores based on the generations of trained generative net against the samples in the test (reference) set.

D. Evaluation Metrics

We include commonly used metrics such as BLEU [12], METEOR score [13] and ROUGE-L score [14]. Since neither of these metrics is designed to measure how well the generated sequences fit the target grammar, we propose two additional metrics to evaluate them. The first one measures the percentage of the generated sequences that are grammatically correct (labeled as SYNTAX). For SQL generation tasks, we additionally report the percentage of generated sequences which obey the schema (labeled as SCHEMA, evaluate the correctness of entity and relation for the generated SQL).

E. Quantitative Results

Table I, Table II and Table III show the quantitative results on synthetic dataset PLD, SQL-A, SQL-B respectively. Firstly, we observe that the tree-based frameworks, including the proposed TreeGAN, achieve 100% SYNTAX correctness regards the pre-defined grammar while other baselines perform much worse. This results show that the proposed TreeGAN could fully capture the given syntax information and generate grammatically correct sequence. Besides, we discover that even though without explicit input of schema, the proposed TreeGAN has higher chance for capturing the underlying semantic pattern, given at least 3.66% and 7.85% improvement on SCHEMA in SQL-A and in SQL-B respectively. More generally, we use three popular NLP metrics to evaluate the quality of the generated sequences. We can clearly see the superiority of TreeGAN, who consistently outperforms the compared methods in terms of the BLEU, METEOR and ROUGE-L (except the case of ROUGE-L in SQL-B, where TreeGAN still obtain competitive results). These results reflect the quality of sequences generated by TreeGAN is better than the generations of compared methods.

F. Qualitative Results

Table V samples several generations on SQL-B dataset for qualitative evaluation. We mainly compare the generations of TextGAN with SeqGAN to demonstrate the advantage of employing tree structure generator and discriminator in GANs on sequence. Consistent with the results we have seen in quantitative evaluation, SeqGAN’s generations could not perfectly follow the underlying grammar and exhibit syntax errors (highlighted in red). As shown in Table V, the generation 4 of SeqGAN mistakenly applies ‘count’ aggregation on a numerical value and does not close the ‘from’ clause correctly. Similar syntax errors can also be observed in the generation 5 and 6 by SeqGAN. Meanwhile, TreeGAN incorporates the pre-defined grammar, and all its generations are valid. We randomly select some examples in Table V. The generation 7 and 8 mimic the ground truth 1 and 2 well and capture the underlying schema correctly. Generation 9 resembles ground truth 3 and extend it with an extra ‘where’ clause.

V. EXPERIMENTS ON REAL DATA

A. Dataset

We test our proposed model on the python code dataset [15] from django project. It is a collection of lines of python code, and each performs a functional task. We use Python AST package and Astor package to construct and parse the AST corresponds to each line of code in the dataset. The code in Django dataset is diverse and spanning a wide variety of real-world use cases such as I/O operations, exception handling, and mathematical computation. We follow the same setting and same evaluation metrics (SYNTAX is not reported due to the freeness of Python grammar) as in the previous section.

1. [http://astor.readthedocs.io/en/latest/]
2. [https://www.djangoproject.com/]

TABLE V: SQL query generation in SQL-B. Syntax errors are highlighted in red color.

| Numbers | Generations |
|---------|-------------|
| 1       | select count(authenticated) from America where alight>3; |
| 2       | select driftpin, min(deject) from Danmark where driftpin=16; |
| 3       | select hedy from Hungary; |
| 4       | select count(17), min(acoustically) from; |
| 5       | select max(cookstove), gainfully, min (), min(buttonhole) from America; |
| 6       | select aalesund from Brazil where hanuman acoustically Hungary; |
| 7       | select min(jacarta) from Jamaica; |
| 8       | select min(endogenous) from Brazil where epigraphical=1; |
| 9       | select hedy from Hungary where deject!=2; |
TABLE VI: Python code generation in Django. Syntax errors are highlighted in red color.

| Code generated by SeqGAN [2] | Code generated by TreeGAN (our method) |
|-------------------------------|---------------------------------------|
| name=self._save, name, content, self | table = connections[db].ops.quote_name(self, table) |
| from django.ImproperlyConfigured | if exp is None or exp > time.time() |
| import 0 | return urljoin(self.base_url, file_path_to_url()) |

B. Quantitative Results

Table IV shows the results quantitative evaluation on Django dataset, from which we discover TreeGAN achieves 6.82% improvement against TreeGen and 18.14% improvement against SeqGAN in terms of BLEU score. As to the METEOR score, TreeGAN improves the performance 1.75% against TreeGen and 2.34% against SeqGAN. We also discover obvious improvement has been made by TreeGAN against TreeGen and SeqGAN in terms of ROUGE-L. Hence, TreeGAN exhibits similar advantages on the real data as in the synthetic study.

C. Qualitative Results

Table VI shows generations from TreeGAN and SeqGAN on Django dataset. Similar to the results obtained in the synthetic study, we found although SeqGAN could mimic the real Python code, it exhibits several types of syntax errors (highlighted in red). Generation 4 indicates SeqGAN sometimes could not correctly fill the function arguments, generation 5 exhibits a misunderstanding of import statement, while generation 6 demonstrates SeqGAN has difficulty in pairing the parentheses. Meanwhile, generation 7 and 8 show the capability of TreeGAN on learning the usage of assignment statement, function call, conditional statement, etc. These observations indicate that on code generation tasks, TreeGAN could effectively plug in complex grammatical rules and generate valid code snippets.

We also discuss the limitation of TreeGAN, which could shed a light on future extension. From generation 9, we can see TreeGAN has difficulty in understanding the concept of inheritance and the member function, where the parent class of ‘META’ does not have the member function called ‘new_file()’, the call is invalid and causes a running-time error. There are about 3.7% of generations by TreeGAN exhibit the similar semantic error in our experiments. It is not difficult to identify that this semantic error in Python is the counterpart of the schema error in SQL. It is possible for our model to learn these semantic pattern from the data, but it may need a better way to guide the learning process for fully capturing the semantic, which could be a future direction for this work.

VI. Conclusion

We proposed a syntax-aware GAN model called TreeGAN for sequence generation. We transform the problem into parse tree generation to incorporate the rich grammar information, and both the generator and discriminator are well-tailored to encode the syntax properly. The experiments on both synthetic datasets and real-world datasets demonstrate that TreeGAN is a promising adversarial learning framework for syntax-aware sequence generation.

REFERENCES

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in Proc. 28th Advances in Neural Information Processing Systems (NIPS’14), 2014, pp. 2672–2680.
[2] L. Yu, W. Zhang, J. Wang, and Y. Yu, “Seqgan: sequence generative adversarial nets with policy gradient,” in Proc. 31st AAAI Conf. on Artificial Intelligence(AAAI’17), 2017, pp. 2852–2858.
[3] Y. Zhang, Z. Gan, K. Fan, Z. Chen, R. Henao, D. Shen, and L. Carin, “Adversarial feature matching for text generation,” in Proc. 34th Int. Conf. Machine Learning (ICML’17), 2017.
[4] Z. Hu, Z. Yang, X. Liang, R. Salakhutdinov, and E. P. Xing, “Toward controlled generation of text,” in Proc. 34th Int. Conf. Machine Learning (ICML’17), 2017, pp. 1587–1596.
[5] P. Yin and G. Neubig, “A syntactic neural model for general-purpose code generation,” in Proc. 57th Annual Meeting of the Association for Computational Linguistics (ACL’17), 2017.
[6] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” in Proc. 4th International Conf. on Learning Representations (ICLR’16), 2016.
[7] N. Chomsky, “Three models for the description of language,” IRE Transactions on information theory, vol. 2, no. 3, pp. 113–124, 1956.
[8] T. Kai, S. Richard, and D. Christopher, “Improved semantic representations from tree-structured long short-term memory networks,” in Proc. 55th Annual Meeting of the Association for Computational Linguistics (ACL’15), 2015.
[9] J. E. Hopcroft, R. Motwani, and J. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2006.
[10] R. Sutton, D. McAllester, S. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” in Proc. 14th Advances in Neural Information Processing Systems (NIPS’00), 2000, pp. 1057–1063.
[11] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.
[12] K. Papineni, S. Roukos, T. Ward, and W. Zhu, “Bleau: a method for automatic evaluation of machine translation,” in Proc. 42nd Annual Meeting of the Association for Computational Linguistics (ACL’02), 2002, pp. 311–318.
[13] M. Denkowski and A. Lavie, “Meteor universal: Language specific translation evaluation for any target language,” in Proc. of the 9th Workshop on Statistical Machine Translation (WMT’14), 2014, pp. 376–380.
[14] C. Lin, “Rouge: A package for automatic evaluation of summaries,” in Proc. of Workshop on Text Summarization Branches Out, Post Conference Workshop of ACL’04, 2004.
[15] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, “Learning to generate pseudo-code from source code using statistical machine translation (t),” in Proc. 30th IEEE/ACM International Conf. Automated Software Engineering (ASE’15), 2015, pp. 574–584.