Enforcing Grammar in Code Synthesis with Transformers

Dmytro Vitel, Stephen Steinle, John Licato
University of South Florida
4202 E Fowler Ave, Tampa, FL 33620
dvitel@usf.edu, ssteinle@usf.edu, licato@usf.edu

Abstract

Even more so than natural language, code is extremely sensitive to syntax; a small error could make an entire snippet invalid. It is therefore important to explore methods for ensuring syntactic correctness in generated code. Existing methods to resolve this issue often rely on the complex architecture of syntax-guided decoders. In this work, we present the grammar enforcement method, which introduces a separate layer that constrains the decisions of the transformer during fine-tuning according to syntactic constructs present both in the target language grammar and the given training set. We experiment with the Hearthstone dataset to study its effects on result programs and compare it with the existing state-of-art syntax-guided decoders. We demonstrate a statistically significant positive effect of grammar enforcement on the quality of generated programs in terms of exact match accuracy and grammatically correct percent of samples. At the same time, we observe lower values for text-based metrics, chrF, and BLEU, potentially indicating their inability to represent the quality of generated abstract syntax sequences.

Introduction

A new generation of transformers, like GPT-3 (Brown et al. 2020) and its code-generating descendant Codex (Chen et al. 2021), demonstrates great ability in text and code synthesis tasks. Even without being explicitly told to learn grammar, the standard processes of pretraining on massive corpora seem to facilitate the learning of syntactic structures. But this may not be sufficient for applications where even one grammatical or structural mistake in the output of the network can constitute failure—e.g., for programming code generation. For this reason, existing work studies introduce formal grammar into the process of token selection during code generation to better guide transformer learning and produce quality programs in the same training time.

In this work, we consider an approach we call the grammar enforcement (GE) method, which adds a separate grammatical layer on top of the existing parallel transformer architecture. This layer does not have any weights to train but constrains the transformer output logits to values that would generate syntactically correct, or near-correct, programs. The layer works after fine-tuning and continues altering decisions at the system production runtime. In the context of this paper, we use the Hearthstone dataset (Vitel 2023b) and the programmatic description of Python grammar as defined in the AST python module (Python Software Foundation 2023).

Related Work

Syntax-guided decoders represent syntactic language models capable of constraining decisions about tokens in place according to grammatical rules and long-distance dependencies in the code. State-of-art models usually incorporate the target language’s grammar into the neural network architecture. We discuss decoders that consider Python code synthesis on the Hearthstone dataset and compare their performance against our method.

LSTM-based RNN approaches to syntax-guided decoders are presented in (Rabinovich, Stern, and Klein 2017) and (Yin and Neubig 2017). A dedicated RNN module is used for a separate grammatical rule of the language and the generation of literals/identifiers. Better performance is demonstrated in (Sun et al. 2019), where authors analyze CNN-based syntax-guided decoder. More recent work (Sun et al. 2020) introduces a complex TreeGen transformer architecture with natural language, abstract syntax, and decoder neural network blocks. Authors also incorporate convolutional layers to combine child node features. The obtained result is used in the abstract syntax tree (AST) level attention. Based on this work, (Zhu et al. 2021) presents the development of the syntax-guided Recorder as the solution to automatic program repair. An alternative approach to grammar usage on the level of transformer attention is demonstrated in (Sartran et al. 2022) and proposes to use stacks to form an attention filter. Recent publication (Zhong et al. 2022) addresses the question of including unit-test information of generated code into the iterative process of decoder application. The form of constrained code synthesis is presented also in the work on API dependency graph embedding (Lyu et al. 2021).

All these approaches incorporate complex architectural decisions with additional weights for tuning. In contrast, our method relies on the black-boxed transformer and applies statistical information from the target grammar and the training set.

Copyright © 2023 by the authors. All rights reserved.
Grammar Enforcement Method

The grammar enforcement method includes two components: the grammar collector and the grammar enforcement layer. The goal of the method is to constrain the underlying transformer logits to values that correspond to the target grammar and given training set, increasing the probability that correct translations are made. The first step in this process is to collect statistics on present syntactical constructs in the training set. Then, during the code generation step, we apply the statistics as a fuzzy mask to encourage feasible labels, where the labels are classification candidates for a particular place in the generated sentence. The source code of the GE layer, statistics collector, fine-tuning, and obtained crude data are publicly available (Vitel 2023a).

Grammar Statistics Collector

We consider a target Hearthstone program in AST form. Every non-leaf tree node corresponds to a non-terminal grammar rule application and tree leaves are terminals. The grammar collector aggregates only the subset of python syntactic constructs which are present in the Hearthstone training set. For example, if a Hearthstone card code does not have list comprehensions, ListComp would not be present in the collected sub-grammar as an expr node. As a result, the model does not have to consider full grammar when choosing what expression should be generated. More precisely the following information is collected:

1. AST node constructor (also referred to as a symbol), a type that allows node creation.
2. AST node group, grammatical category, such as stmt or expr, to which symbol belongs.
3. A symbol’s attribute list describing children of the node:
   (a) The grammatical group of a child.
   (b) Grammar modifiers (such as ? or *) denote a corresponding number of symbol instances in place according to grammar and training set.
   (c) Primitive type of a literal (if the child is the one).
   (d) Possible subset of symbols of the grammatical group which were met in the training set.

Target program representation

We transform the Hearthstone target programs of training, dev, and test sets to ASTs with the data preprocessing function. Then, formed trees are linearized by converting each symbol into a corresponding token sequence with preorder traversal. This allows the transformer to generate the abstract syntax. The symbol-to-token mapping is handpicked according to the underlying transformer vocabulary to simplify the final representation. Symbol names of the grammar correspond to at most one token to minimize the number of decisions the transformer should make to start node generation. This currently requires the knowledge of the black-boxed tokenizer, but in the future could be lifted with a more complex mapping of symbols to many tokens. The ‘Argument’ symbol is an example of a many-token symbol name, where 2 logits of the transformer have to decide the start of the text span for the Argument AST node.

The following snippet demonstrates the example of the tokenized form of AST for the class corresponding to the “Hellfire” card:

```
module class Hellfire  
  SpellCard load  
  __init__ arguments ...
```

Tree linearization requires utility tokens (NEND token `''`, LST token '<[', LitStart token '||') to demarcate the end of a node or list. The post-processing procedure ensures that the python code can be unambiguously recovered from the AST string if it is grammatically correct.

```
NEND``
```
NEND``
```
GenLst

Figure 1: GE layer as a transducer.

Grammar Enforcement Layer

The GE layer implements the finite state transducer (Figure 1). For each generated program token position, the layer outputs a set of filter values for underlying transformer logits. The filter values are calculated from the grammar collector and decisions for previous positions of the generated sample.

If the GE layer allows a vocabulary token to be a legitimate choice at the current position, the filter value associated with the logit token is assigned the value of the up level system hyperparameter. Otherwise, the assigned value is taken from the down level hyperparameter. During experimentation, we set the up level to be 1.0, meaning allowed logits are passed from the transformer to the system output unmodified. The down level varies from -0.2 to 1.2. The GE layer is functionally disabled by setting all values to 1.0, meaning that penalized transformer logits are not modified, while the level of 0.0 fully enables the GE layer and masks logits for grammatically incorrect labels.

The GE layer always starts generating `Module` symbol. Therefore, the first position of every sample has the up-level set only for the ‘module’ token logit. Other logits at this place have the down-level filter value. When the layer enters the synthesis of expression, it allows more alternatives for the position. Figure 1 demonstrates three states of the layer.

The synthesis of AST node is denoted by ‘GenSym’. The GE layer computes possible symbols based on the expected
Our goal is to establish the effect of grammar enforcement all 912 tokens, we discard the rest of the transformer logits. To accomplish this, the GE layer finishes the AST string and does not use plain logits before softmax and cross-entropy loss are applied. If the GE layer transitions back to 'GenSym' for the next parent’s attribute, the layer constrains possible tokens for plain text when a symbol is picked according to transformer logits altered by a scale factor for up-level value. For example, if the predicted symbol causes an additional recursive tree level, its depth \( d \) will define the penalty and skew choice toward recursive termination. The depthPenalty varies from 0.98 to 1.0 (Table 1). Similarly, lenPenalty introduces a bias towards shorter identifiers or children lists. The final filter value is defined as follows (\( i \) denotes the token or child position in the sequence):

\[
\text{filter}_{d,i} = \text{depthPenalty}^{d-1} \ast \text{lenPenalty}^{i-1} \ast \text{upLevel}
\]

On the training set, we apply teacher forcing inside GE layer. The correct label is used in deciding the next GE state and filter values. The prediction is used in the GE layer when the contextual symbols correspond to the generation of one child with the following retry of finalization.

The ‘GenLit’ state starts the generation of the plain text of a literal or identifier. When NEND ends the text span, the GE layer proceeds to the next parent’s attribute, while the contextual symbols correspond to the generation of one child with the following retry of finalization.

The state ‘GenLst’ denotes the synthesis of AST node children. The GE layer allows the selection of context-appropriate symbols or the NEND. Both selections lead to ‘GenSym’, but the NEND finalizes the children’s list, and the GE layer proceeds to the next parent’s attribute, while the contextual symbols correspond to the generation of one child.

To avoid long token sequences for identifiers, literals, and deep trees, we introduce depth and length penalties as a scale factor for up-level value. For example, if the predicted symbol causes an additional recursive tree level, its depth \( d \) will define the penalty and skew choice toward recursive termination. The depthPenalty varies from 0.98 to 1.0 (Table 1). Similarly, lenPenalty introduces a bias towards shorter identifiers or children lists. The final filter value is defined as follows (\( i \) denotes the token or child position in the sequence):

\[
\text{filter}_{d,i} = \text{depthPenalty}^{d-1} \ast \text{lenPenalty}^{i-1} \ast \text{upLevel}
\]

On the training set, we apply teacher forcing inside GE layer. The correct label is used in deciding the next GE state and filter values. The prediction is used in the GE layer when performance is evaluated on test set.

The final GE filter tensor is multiplied onto transformer logits before softmax and cross-entropy loss are applied. If the GE layer finishes AST string and does not use all 912 tokens, we discard the rest of the transformer logits.

**Experiment**

Our goal is to establish the effect of grammar enforcement on fine-tuning the underlying transformer based on the quality of generated programs. The down-level hyperparameter was introduced to vary the GE effect in the range from fully disabled GE to fully enabled and beyond. Table 1 shows experimental system parameters.

We picked distilgpt2 (HuggingFace 2019) as the underlying transformer. It is an English-language model pre-trained with the supervision of the smallest version of GPT-2 (Rafford et al. 2019). We fine-tune distilgpt2 with GE on the Hearststone dataset. During training, the system receives a target and source sentence separated by a special token.

We use the following metrics to compare generated code.

**Exact match %** is the percent of generated token sequences that exactly match gold AST sequences from training/dev/test set.

**Correct %** is the percent of grammatically correct programs. We say that a generated AST string is grammatically correct when it is possible to post-process it back to AST form and then unparse it to a python program.

**Program length** is the number of tokens corresponding to generated complete program. It excludes a tail of non-used tokens. (HuggingFace 2023) (Microsoft 2023)

We ran the system for different random seeds and down levels (Table 1). Table 2 summarizes metrics for GE down level 0 (enabled), 0.5 (partially enabled), and 1 (disabled) on the test set after fine-tuning. The GE layer introduces statistically significant differences in performance. Exact match accuracy and correct % are improved with GE. Enabling the GE layer drops differences in performance. Exact match accuracy and cross-entropy loss are text-based metrics commonly used in comparing text. (HuggingFace 2023) (Microsoft 2023)

**Experiment**

Our goal is to establish the effect of grammar enforcement on fine-tuning the underlying transformer based on the quality of generated programs. The down-level hyperparameter
Figure 2: Generated program quality. The percentage of exactly matched programs to gold label programs, grammatically correct programs, complete but incorrect, and incomplete programs.

we go beyond the down-level of 1., we penalize grammatically correct logits and, as expected, observe a further drop in quality. Decreasing the down level below zero only affects the length and depth penalty (makes them more extensive) but degrades the exact match slightly.

Figure 3: The percent of exactly matched programs from the test set for different GE layer strengths. A down level of 0 corresponds to a fully enabled GE, while the value of 1 is the GE layer fully disabled. Results show the mean value of the metric and error bars for standard deviation.

It is interesting to observe that text-based metrics chrF and BLEU have the opposite effect—they seem to improve without the GE layer. These metrics are based on n-gram models (character- or word-based) and consider the n-gram overlaps in the gold and generated programs. Because the system without the GE layer generally produces longer programs, we assume that these n-gram overlaps could explain the increase in text-based metrics. At the same time, the general syntactic structure of the program could be incorrect and even incomplete in 912 tokens.

Table 3: Performance of different syntax-guided decoders. StrAcc is an exact match percent, Acc+ allows errors in class and variable names, but programs are still correct.

Analysis of system mistakes shows that the transformer with GE still could generate wrong multi-token identifiers. To fix the issue, our future work is to improve the GE method with techniques of name-stripping preprocessing and implementation of static semantics analysis.

Table 3 compares our method to described earlier syntax-guided decoders. It reports exact match percent (StrAcc), percent with identifiers mismatch only (Acc), and BLEU metrics. The best model TreeGen-B demonstrates 31.8% exact match accuracy. For our GE method, we present the best exact match 30.3%, averaged over random seeds (down = 0.7). Notice, however, that several fine-tuning runs demonstrate the test set exact match above 30%. The maximum obtained value, 33.3%, is observed, for example, for random seed 313 and down = 0.5. In addition, we only train for 200 epochs, while some mentioned published methods did not provide this parameter at all.

Conclusion

This work presents a grammar enforcement method on top of transformer fine-tuning for code generation, using the Hearthstone dataset as our benchmark. The proposed approach collects statistics of syntactic constructs present in the training set to apply this knowledge in decisions of the enforcement layer. The result of enforcement is the filter tensor from the GE transducer, the multiplication of which onto transformer logits constrains the system output according to grammar rules.

Collected metrics show the positive effect of the GE method on the quality of generated programs. The percent of matched to gold programs increases from 24% to 30%, and the number of grammatically correct samples increases from 33% to 100%. It makes the GE layer attractive in applications requiring grammatically feasible solutions. In addition, the GE layer introduces length and depth-based pressure, which skews search toward shorter programs, and, at the same time, eliminates incomplete in the maximum number of tokens samples.

Although our method does not claim to demonstrate state-of-the-art results when compared to much more powerful systems like GPT-3, we argue that it is worthwhile to continue exploring whether enforcement of grammatical rules in language model output within the layers themselves can yield results on smaller, more computationally tractable models.