Retrieval-Based Neural Code Generation

Shirley Anugrah Hayati **Raphael Olivier** **Pravalika Avvaru**
Pengcheng Yin **Anthony Tomasic** **Graham Neubig**

Language Technologies Institute, Carnegie Mellon University
{shayati,rolivier,pavvaru,pcyin,tomasic,gneubig}@cs.cmu.edu

Abstract

In models to generate program source code from natural language, representing this code in a tree structure has been a common approach. However, existing methods often fail to generate complex code correctly due to a lack of ability to memorize large and complex structures. We introduce ReCode, a method based on subtree retrieval that makes it possible to explicitly reference existing code examples within a neural code generation model. First, we retrieve sentences that are similar to input sentences using a dynamic-programming-based sentence similarity scoring method. Next, we extract n-grams of action sequences that build the associated abstract syntax tree. Finally, we increase the probability of actions that cause the retrieved n-gram action subtree to be in the predicted code. We show that our approach improves the performance on two code generation tasks by up to +2.6 BLEU.[1]

1 Introduction

Natural language to code generation, a subtask of semantic parsing, is the problem of converting natural language (NL) descriptions to code (Ling et al., 2016; Yin and Neubig, 2017; Rabinovich et al., 2017). This task is challenging because it has a well-defined structured output and the input structure and output structure are in different forms.

A number of neural network approaches have been proposed to solve this task. Sequential approaches (Ling et al., 2016; Jia and Liang, 2016; Locascio et al., 2016) convert the target code into a sequence of symbols and apply a sequence-to-sequence model, but this approach does not ensure that the output will be syntactically correct. Tree-based approaches (Yin and Neubig, 2017; Rabinovich et al., 2017) represent code as Abstract Syntax Trees (ASTs), which has proven effective in improving accuracy as it enforces the well-formedness of the output code. However, representing code as a tree is not a trivial task, as the number of nodes in the tree often greatly exceeds the length of the NL description. As a result, tree-based approaches are often incapable of generating correct code for phrases in the corresponding NL description that have low frequency in the training data.

In machine translation (MT) problems (Zhang et al., 2018; Gu et al., 2018; Amin Farajian et al., 2017; Li et al., 2018), hybrid methods combining retrieval of salient examples and neural models have proven successful in dealing with rare words. Following the intuition of these models, we hypothesize that our model can benefit from querying pairs of NL descriptions and AST structures from training data.

In this paper, we propose ReCode, and adaptation of Zhang et al. (2018)'s retrieval-based approach neural MT method to the code generation problem by expanding it to apply to generation of tree structures. Our main contribution is to introduce the use of retrieval methods in neural code generation models. We also propose a dynamic programming-based sentence-to-sentence alignment method that can be applied to similar sentences to perform word substitution and enable retrieval of imperfect matches. These contributions allow us to improve on previous state-of-the-art results.

2 Syntactic Code Generation

Given an NL description q, our purpose is to generate code (e.g. Python) represented as an AST a. In this work, we start with the syntactic code gen-

[1] Code available at https://github.com/sweetpeach/ReCode

eration model by Yin and Neubig (2017), which uses sequences of actions to generate the AST before converting it to surface code. Formally, we want to find the best generated AST $\hat{a}$ given by:

$$\hat{a} = \arg\max_{a} p(a|q)$$

$$p(a|q) = \prod_{t=1}^{T} p(y_t|y_{<t}, q)$$

where y_t is the action taken at time step t and $y_{<t} = y_1 ... y_{t-1}$ and T is the number of total time steps of the whole action sequence resulting in AST a.

We have two types of actions to build an AST: APPLYRULE and GENTOKEN. APPLYRULE(r) expands the current node in the tree by applying production rule r from the abstract syntax grammar[2] to the current node. GENTOKEN(v) populates terminal nodes with the variable v which can be generated from vocabulary or by COPYing variable names or values from the NL description. The generation process follows a preorder traversal starting with the `root` node. Figure 1 shows an action tree for the example code: the nodes correspond to actions per time step in the construction of the AST.

Interested readers can reference Yin and Neubig (2017) for more detail of the neural model, which consists of a bidirectional LSTM (Hochreiter and Schmidhuber, 1997) encoder-decoder with action embeddings, context vectors, parent feeding, and a copy mechanism using pointer networks.

3 RECODE: Retrieval-Based Neural Code Generation

We propose RECODE, a method for retrieval-based neural syntactic code generation, using retrieved action subtrees. Following Zhang et al. (2018)'s method for neural machine translation, these retrieved subtrees act as templates that bias the generation of output code. Our pipeline at test time is as follows:

- retrieve from the training set NL descriptions that are most similar with our input sentence (section 3.1),
- extract **n-gram action subtrees** from these retrieved sentences' corresponding target ASTs (section 3.2),
- alter the copying actions in these subtrees, by substituting words of the retrieved sentence with corresponding words in the input sentence (section 3.3), and
- at every decoding step, increase the probability of actions that would lead to having these subtrees in the produced tree (section 3.4).

3.1 Retrieval of Training Instances

For every retrieved NL description q_m from training set (or *retrieved sentence* for short), we compute its similarity with input q, using a sentence similarity formula (Gu et al., 2016; Zhang et al., 2018):

$$\text{sim}(q, q_m) = 1 - \frac{d(q, q_m)}{\max(|q|, |q_m|)}$$

where d is the edit distance. We retrieve only the top M sentences according to this metric where M is a hyperparameter. These scores will later be used to increase action probabilities accordingly.

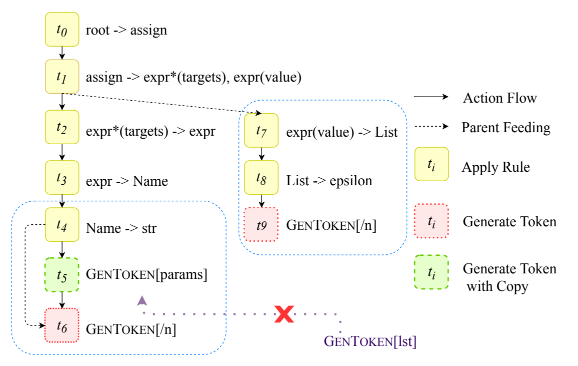
3.2 Extracting N-gram Action Subtrees

In Zhang et al. (2018), they collect n-grams from the output side of the retrieved sentences and encourage the model to generate these n-grams. Word n-grams are obvious candidates when generating a sequence of words as output, as in NMT. However, in syntax-based code generation, the generation target is ASTs with no obvious linear structure. To resolve this problem, we instead use *retrieved pieces* of n-gram subtrees from the target code corresponding to the retrieved NL descriptions. Though we could select successive nodes in the AST as retrieved pieces, such as `[assign; expr*(targets); expr]` from Figure 1, we would miss important structural information from the rules that are used. Thus, we choose to exploit actions in the generation model rather than AST nodes themselves to be candidates for our retrieved pieces.

In the action tree (Figure 1), we considered only successive actions, such as subtrees where each node has one or no children, to avoid overly rigid structures or combinatorial explosion of the number of retrieved pieces the model has to consider. For example, such an action subtree would be given by `[assign → expr*(targets), expr(value) ; expr(value) → List; List → epsilon]`.

As the node in the action tree holds structural information about its children, we set the subtrees

[2]https://docs.python.org/2/library/ast.html



Input : params *is an empty list* **Target Code** : `params = [ ]`

Retrieved: `List lst` *is an empty list* **Retrieved Code:** `lst = [ ]`

Figure 1: The action sequence used to generate AST for the target code given the input example. Dashed nodes represent terminals. Each node is labeled with time steps. APPLYRULE action is represented as rule in this figure. Blue dotted boxes denote 3-gram action subtrees. Italic words are unedited words. Red bold words are different object names.

to have a fixed depth, linear in the size of the tree. These can be considered "n-grams of actions", emphasizing the comparison with machine translation which uses n-grams of words. n is a hyperparameter to be tuned.

3.3 Word Substitution in Copy Actions

Using the retrieved subtree without modification is problematic if it contains at least one node corresponding to a COPY action because copied tokens from the retrieved sentence may be different from those in the input. Figure 1 shows an example when the input and retrieved sentence have four common words, but the object names are different. The extracted action n-gram would contain the rule that copies the second word ("lst") of the retrieved sentence while we want to copy the first word ("params") from the input.

By computing word-based edit distance between the input description and the retrieved sentence, we implement a one-to-one sentence alignment method that infers correspondences between uncommon words. For unaligned words, we alter all COPY rules in the extracted n-grams to copy tokens by their aligned counterpart, such as replace "params" with "lst", and delete the n-gram subtree, as it is not likely to be relevant in the predicted tree. Thus, in the example in Figure 1, the GENTOKEN(LST) action in t_5 will not be executed.

3.4 Retrieval-Guided Code Generation

N-gram subtrees from all retrieved sentences are assigned a score, based on the best similarity score of all instances where they appeared. We normalize the scores for each input sentence by subtracting the average over the training dataset.

At decoding time, incorporate these retrieval-derived scores into beam search: for a given time step, all actions that would result in one of the retrieved n-grams u to be in the prediction tree has its log probability $\log(p(y_t \mid y_1^{t-1}))$ increased by $\lambda * \text{score}(u)$ where λ is a hyperparameter, and $\text{score}(u)$ is the maximal $\text{sim}(q, q_m)$ from which u is extracted. The probability distribution is then renormalized.

Dataset	HS	Django
Train	533	16,000
Dev	66	1,000
Test	66	1,805
Avg. tokens in description	39.1	14.3
Avg. number of nodes of AST	136.6	17.2

Table 1: Dataset statistics as reported Yin and Neubig (2017)

4 Datasets and Evaluation Metrics

We evaluate RECODE with the Hearthstone (HS) (Ling et al., 2016) and Django (Oda et al., 2015) datasets, as preprocessed by Yin and Neubig (2017). HS consists of Python classes that implement Hearthstone card descriptions while Django contains pairs of Python source code and English pseudo-code from Django web framework. Table 1 summarizes dataset statistics.

For evaluation metrics, we use accuracy of exact match and the BLEU score following Yin and Neubig (2017).

5 Experiments

For the neural code generation model, we use the settings explained in Yin and Neubig (2017). For the retrieval method, we tuned hyperparameters and achieved best result when we set $n_{max} = 4$ and $\lambda = 3$ for both datasets[3]. For HS, we set $M = 3$ and $M = 10$ for Django.

We compare our model with Yin and Neubig (2017)'s model that we call YN17 for brevity, and a sequence-to-sequence (SEQ2SEQ) model that we implemented. SEQ2SEQ is an attention-enabled encoder-decoder model (Bahdanau et al., 2015). The encoder is a bidirectional LSTM and the decoder is an LSTM.

5.1 Results

Table 2 shows that RECODE outperforms the baselines in both BLEU and accuracy, providing ev-

[3]n-gram subtrees are collected up to n_{max}-gram

idence for the effectiveness of incorporating retrieval methods into tree-based approaches.

	HS		Django	
	Acc	**BLEU**	**Acc**	**BLEU**
SEQ2SEQ	0.0	55.0	13.9	67.3
YN17	16.2	75.8	71.6	84.5
ASN[†]	18.2	77.6	-	-
ASN + SUPATT[†]	*22.7*	*79.2*	-	-
RECODE	**19.6**	**78.4**	**72.8**	**84.7**

Table 2: Results compared to baselines. YN17 result is taken from Yin and Neubig (2017). ASN result is taken from Rabinovich et al. (2017)

.

We ran statistical significance tests for RECODE and YN17, using bootstrap resampling with $N =$ 10,000. For the BLEU scores of both datasets, $p <$ 0.001. For the exact match accuracy, $p < 0.001$ for Django dataset, but for Hearthstone, $p > 0.3$, showing that the retrieval-based model is on par with YN17. It is worth noting, though, that HS consists of long and complex code, and that generating exact matches is very difficult, making exact match accuracy a less reliable metric.

We also compare RECODE with Rabinovich et al. (2017)'s Abstract Syntax Networks with supervision (ASN+SUPATT) which is the state-of-the-art system for HS. RECODE exceeds ASN without extra supervision though ASN+SUPATT has a slightly better result. However, ASN+SUPATT is trained with supervised attention extracted through heuristic exact word matches while our attention is unsupervised.

5.2 Discussion and Analysis

From our observation and as mentioned in Rabinovich et al. (2017), HS contains classes with similar structure, so the code generation task could be simply matching the tree structure and filling the terminal tokens with correct variables and values. However, when the code consists of complex logic, partial implementation errors occur, leading to low exact match accuracy (Yin and Neubig, 2017). Analyzing our result, we find this intuition to be true not only for HS but also for Django.

Examining the generated output for the Django dataset in Table 3, we can see that in the first example, our retrieval model can successfully generate the correct code when YN17 fails. This difference suggests that our retrieval model benefits from the action subtrees from the retrieved sentences. In the second example, although our generated code does not perfectly match the reference code, it has a higher BLEU score compared

Example 1	
"if offset is lesser than integer 0, sign is set to '-', otherwise sign is '+' "	**Input**
`sign = offset < 0 or '-'`	**YN17**
`sign = '-' if offset < 0 else '+'`	**RECODE**
`sign = '-' if offset < 0 else '+'`	**Gold**
Example 2	
"evaluate the function timesince with d, now and reversed set to boolean true as arguments, return the result."	**Input**
`return reversed(d, reversed=now)`	**YN17**
`return timesince(d, now, reversed=now)`	**RECODE**
`return timesince(d, now, reversed=True)`	**Gold**
Example 3	
"return an instance of SafeText , created with an argument s converted into a string ."	**Input**
`return SafeText(bool(s))`	**YN17**
`return SafeText(s)`	**RECODE**
`return SafeString(str(s))`	**Gold**

Table 3: Django examples on correct code and predicted code with retrieval (RECODE) and without retrieval (YN17).

NAME_BEGIN Earth Elemental NAME_END ATK_BEGIN 7 ATK_END DEF_BEGIN 8 DEF_END COST_BEGIN 5 COST_END DUR_BEGIN -1 DUR_END TYPE_BEGIN Minion TYPE_END PLAYER_CLS_BEGIN Shaman PLAYER_CLS_END RACE_BEGIN NIL RACE_END RARITY_BEGIN Epic RARITY_END DESC_BEGIN Taunt . Overload : (3) DESC_END. **Input**

```
class EarthElemental (MinionCard) :
    def __init__ (self) :
        super ( ).__init__ ("Earth Elemental", 5,
          CHARACTER_CLASS.SHAMAN, CARD_RARITY.EPIC,
          buffs=[Buff(ManaChange(Count
        (MinionSelector(None, BothPlayer())), -1))])
    def create_minion (self, player) :
        return Minion(7, 8, taunt=True)
```
YN17

```
class EarthElemental (MinionCard) :
    def __init__ (self) :
        super ( ).__init__ ("Earth Elemental", 5,
          CHARACTER_CLASS.SHAMAN, CARD_RARITY.EPIC,
          overload=3)
    def create_minion (self, player) :
        return Minion(7, 8, taunt=True)
```
RECODE

```
class EarthElemental (MinionCard) :
    def __init__ (self) :
        super ( ).__init__ ("Earth Elemental", 5,
          CHARACTER_CLASS.SHAMAN, CARD_RARITY.EPIC,
          overload=1)
    def create_minion (self, player) :
        return Minion(7, 8, taunt=True)
```
Gold

Table 4: HS examples on correct code and predicted code with retrieval (RECODE) and without retrieval (YN17).

to the output of YN17 because our model can predict part of the code (`timesince(d, now, reversed)`) correctly. The third example shows where our method fails to apply the correct action as it cannot cast `s` to `str` type while YN17 can at least cast `s` into a type (`bool`). Another common type of error that we found RECODE's generated outputs is incorrect variable copying, similarly to what is discussed in Yin and Neubig (2017) and Rabinovich et al. (2017).

Table 4 presents a result on the HS dataset[4]. We can see that our retrieval model can handle complex code more effectively.

6 Related Work

Several works on code generation focus on domain specific languages (Raza et al., 2015; Kushman and Barzilay, 2013). For general purpose code generation, some data-driven work has been

[4]More example of HS code is provided in the supplementary material.

done for predicting input parsers (Lei et al., 2013) or a set of relevant methods (Raghothaman et al., 2016). Some attempts using neural networks have used sequence-to-sequence models (Ling et al., 2016) or tree-based architectures (Dong and Lapata, 2016; Alvarez-Melis and Jaakkola, 2017). Ling et al. (2016); Jia and Liang (2016); Locascio et al. (2016) treat semantic parsing as a sequence generation task by linearizing trees. The closest work to ours are Yin and Neubig (2017) and Rabinovich et al. (2017) which represent code as an AST. Another close work is Dong and Lapata (2018), which uses a two-staged structure-aware neural architecture. They initially generate a low-level sketch and then fill in the missing information using the NL and the sketch.

Recent works on retrieval-guided neural machine translation have been presented by Gu et al. (2018); Amin Farajian et al. (2017); Li et al. (2018); Zhang et al. (2018). Gu et al. (2018) use the retrieved sentence pairs as extra inputs to the NMT model. Zhang et al. (2018) employ a simpler and faster retrieval method to guide neural MT where translation pieces are n-grams from retrieved target sentences. We modify Zhang et al. (2018)'s method from textual n-grams to n-grams over subtrees to exploit the code structural similarity, and propose methods to deal with complex statements and rare words.

In addition, some previous works have used subtrees in structured prediction tasks. For example, Galley et al. (2006) used them in syntax-based translation models. In Galley et al. (2006), subtrees of the input sentence's parse tree are associated with corresponding words in the output sentence.

7 Conclusion

We proposed an action subtree retrieval method at test time on top of an AST-driven neural model for generating general-purpose code. The predicted surface code is syntactically correct, and the retrieval component improves the performance of a previously state-of-the-art model. Our successful result suggests that our idea of retrieval-based generation can be potentially applied to other tree-structured prediction tasks.

Acknowledgements

We are grateful to Lucile Callebert for insightful discussions, Aldrian Obaja Muis for helpful input on early version writing, and anonymous reviewers for useful feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1815287.

References

D. Alvarez-Melis and T. Jaakkola. 2017. Tree structured decoding with doubly recurrent neural networks. In *International Conference on Learning Representations (ICLR)*.

M Amin Farajian, Marco Turchi, Matteo Negri, Marcello Federico, and Fondazione Bruno Kessler. 2017. Multi-Domain Neural Machine Translation through Unsupervised Adaptation. In *Proceedings of the Second Conference on Machine Translation (WMT), Volume 1: Research Papers*, pages 127–137.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 33–34.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 731–742.

Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. 2006. Scalable inference and training of context-rich syntactic translation models. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 961–968.

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

Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor O. K. Li. 2018. Search engine guided neural machine translation. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*.

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

R. Jia and P. Liang. 2016. Data recombination for neural semantic parsing. In *The 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 12–22.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings*, pages 826–836.

Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. 2013. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1294–1303.

Xiaoqing Li, Jiajun Zhang, and Chengqing Zong. 2018. One Sentence One Model for Neural Machine Translation. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

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

Nicholas Locascio, Karthik Narasimhan, Eduardo De Leon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1918–1923.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584.

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

Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 357–367.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 792–800.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 440–450.

Jingyi Zhang, Masao Utiyama, Eiichiro Sumita, Graham Neubig, and Satoshi Nakamura. 2018. Guiding neural machine translation with retrieved translation pieces. In *Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*.