Program Synthesis: Does Feedback Help?

Varun Jain*, Harsh Patel*, Shivam Sahni*, Praveen Venkatesh*, Mrinal Anand, Mayank Singh
Indian Institute of Technology Gandhinagar, India

ABSTRACT

Computers are devices that execute precise instructions provided to them using various programming languages. However, the idea of delivering instructions to a computer through natural language could vastly simplify the act of *programming* as a specific task. Generating code from high-level descriptions for a given program is a significantly challenging task and has been an active area of research in the natural language processing domain. In this paper, we present a novel feedback-based deep learning approach for synthesizing code from human-specified descriptions. Inspired by the dual-learning mechanism, our framework uses a feedback loss to produce more consistent and robust predictions. We show how our approach fares empirically on standard code generation datasets and achieves state-of-the-art results on the NAPS (Natural Program Synthesis) dataset.

ACM Reference Format:
Varun Jain*, Harsh Patel*, Shivam Sahni*, Praveen Venkatesh*, Mrinal Anand, Mayank Singh . 2022. **Program Synthesis: Does Feedback Help?**. In *5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD) (CODS-COMAD 2022), January 8–10, 2022, Bangalore, India.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3493700.3493756

1 INTRODUCTION

The need for high-quality code that adheres to a set of specified rules is increasing every day, with more than 28.7 million developers expected to form part of the development workforce by 2024 [3]. A large part of a developer's workday is spent on writing code to achieve tasks that have already been used in a different context before. This implies that the developer spends time searching for prior solutions for a problem, which is often an arduous process. Program synthesis is the task of generating a program that most likely fits a given specification (provided by the user) of said program. The verbosity of the user specification and the complexity of generating such programs may vary widely depending on the user's application and level of expertise. Early attempts at program synthesis are focused on search-based approaches, which try to find the best match for a given specification via input/output pairs [1, 6, 7]. However, variations in writing style, language, expertise may lead to incorrect inferences made by the synthesis method, leading to ambiguities in the generated code. More modern approaches have resorted to deep learning-based techniques [4, 10] that attempt to capture contextual clues provided in the input specification, resolving conflicts between different writing styles. However, even with advancements in natural language understanding, the task of program synthesis remains unsolved due to: 1) *Very long-range dependencies* : code referenced in a file may have been used in files elsewhere. Contextual clues when writing code lie not only within the file being edited but several files in the file directory, often spanning several million lines in large codebases. 2) *Lack of high-quality, large, real-life datasets* : Most datasets available for program synthesis are synthetic or crowdsourced. This leads to lot of noise in the data that eventually impairs performance.

Our work takes inspiration from the concept of feedback. We attempt to guide the model to learn better and quicker by providing an alternate feedback path apart from a single loss based gradient-descent. We leverage the idea of dual learning [2, 9] which utilizes a two stage reinforcement learning framework based on Seq2Seq networks for language-to-language translation. Our work focuses on a supervised training framework for description to code synthesis and utilizes an additional feedback similarity loss compared between contextual embeddings of an *inferred code specification*, and a *ground truth code specification*.

2 DATASET

1) NAPS Dataset: The NAPS dataset contains 16,410 code sequences and 300 synthetically generated descriptions for each code sequence [10]. The code sequences are in Universal Abstract Syntax Tree (UAST). Since the UAST trees can be converted to other programming languages like Java and C++, the dataset is highly generalisable. The dataset also provides test cases used to compute the accuracy on the dataset. This dataset is highly complex, indicated by its poor performance on standard Seq2Seq models [10], and due to its large program lengths.

2) Algolisp Dataset: The Algolisp dataset contains 79,214 descriptions along with code sequence in a tree format [4]. The code sequences are written in a LISP like programming language. Akin to NAPS, the dataset has test cases that are used to compute the accuracy. We observe that only 89% of the code sequences in the dataset pass their corresponding test cases provided. As a result, as an additional metric, we also compute the *exact match* accuracy which validates whether the output code is an exact match with the ground truth.

3 OUR APPROACH

Recent works that use Seq2Seq and Seq2Tree models have achieved good results in most NLP problems. However, for program synthesis on the NAPS dataset, the Seq2Tree model currently tops the leaderboard with a mere accuracy of 8.8% [10]. This poor performance demonstrates a failure to interpret input code specifications in a reasonable manner.

In our framework, we use attention-based transformer models [8] which are capable of handling long-range dependencies of

*The first 4 authors contributed equally, and are undergraduate students. The authors can be contacted at: {varun.jain, harsh.patel, shivam.sahni, praveen.venkatesh, mrinal.anand, singh.mayank}@iitgn.ac.in .

CODS-COMAD 2022, January 8–10, 2022, Bangalore, India

ACM ISBN 978-1-4503-8582-4/22/01.
https://doi.org/10.1145/3493700.3493756

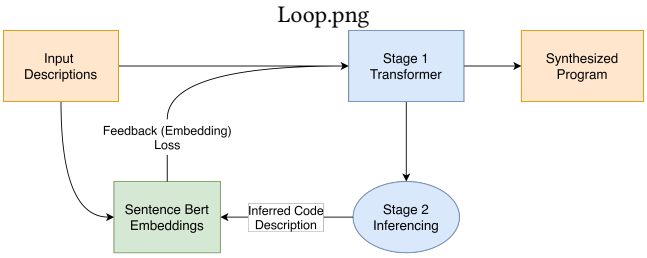

Figure 1: Our Architecture

languages, to enhance the efficiency of this sequence-to-sequence generation task. Figure 1 shows the two stages of our pipeline to synthesize code from a given code specification.

The primary task of the stage-1 transformer is to translate a given high-level code description to the desired code sequence. Stage-2 consists of a pre-trained transformer (trained on the inverted dataset : code to description) that converts predicted code sequences of stage-1 back into a description. We call this prediction as the *inferred description* since it captures the context of the generated program and is inferred as the data passes through the model.

We hypothesize that if the *inferred descriptions* are similar to the input descriptions, the penultimate task of code generation, i.e., the inferences of stage-1, are accurate (close to ground truth). Thus, we formulate a network architecture (fig 1) where the second stage provides feedback to the first stage via a cosine-similarity loss computed between the Sentence-Bert [5] embeddings of the *inferred code descriptions* and the *ground-truth code descriptions*.

We formulate the loss function for training the network as $L_{new} = L_{CE}(\hat{y}, y) + \lambda L_{cosine}(\hat{y}', \hat{x})$, where L is the loss function, λ is a tunable parameter, y is the ground truth code, $\hat{y}$ is the prediction from stage-1, $\hat{y}'$ is the embedding of the prediction from S-BERT, $\hat{x}$ is the ground truth embedding from S-BERT, L_{CE} is the cross entropy stage-1 loss function, and L_{cosine} is a cosine similarity loss computed between the S-BERT embeddings of the *inferred descriptions* and the ground truth descriptions.

Since the vanilla S-BERT model has been pre-trained on a different corpus not meant for code synthesis, we fine-tune the network specifically for our task using the training data.

4 RESULTS

Figure 2 and 3 show a summary of the results obtained by testing our framework on Algolisp and NAPS datasets, respectively.

For the Algolisp dataset, we compare our results with a vanilla sequence to sequence transformer. We also apply the back-translation technique to augment the training set for improving the performance. As seen in the table, our approach outperforms other previously mentioned methods.

For the NAPS dataset, we compare our results with the baseline Seq2Tree model [10] both with and without using out-of-vocabulary (OOV) words. We achieve state-of-the-art results for the NAPS dataset, which shows that the feedback loop provides a better environment for robust training of the principal transformer model, ensuring that the semantic structure of the code is well-understood. With this work, we present a significant base for further study on the problem of program synthesis.

Model | Strategy	Accuracy (% exact match vs steps)			
	10k	*20k*	*40k*	*60k*
Transformer | Vanilla	79.31	93.1	95.08	95.54
Transformer | Back translation	76.61	93.13	94.42	95.33
Transformer | Ours	**88.97**	**94.74**	**95.88**	**95.99**

Figure 2: Results on the Algolisp dataset

Model | Strategy	Accuracy (%) All cases passed	Accuracy (% all cases passed vs steps)		
		10k	20k	200k
Seq2Tree | Without OOV [10]	8.8	-	-	-
Seq2Tree | With OOV [10]	7.9	-	-	-
Transformer | Ours	**55.36**	23.64	40.23	55.36

Figure 3: Results on the NAPS dataset

5 CONCLUSION & FUTURE WORK

In recent years, there has been tremendous progress in the generation of neural programs. However, state-of-the-art models continue to struggle with producing programs with a higher word count because they do not understand the semantics of the generated code. With our feedback-based pipeline, we leverage the multi-task nature of this code generation and code summarization. We show that the model learns and interprets the code's semantics more accurately through this dual learning framework, showing that feedback does indeed help.

In the future, we plan to train the model on multiple languages, forcing the network to learn language agnostic concepts (such as OOP), rather than learn a combined representation of semantics and syntax of a domain-specific language. This can potentially give rise to a more complex understanding of the act of *programming*, leading to increased accuracy of program synthesis.

REFERENCES

[1] Alan W. Biermann. 1978. The Inference of Regular LISP Programs from Examples. *IEEE Transactions on Systems, Man, and Cybernetics* 8, 8 (1978), 585–600. https://doi.org/10.1109/TSMC.1978.4310035

[2] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. 2016. Dual Learning for Machine Translation. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/5b69b9cb83065d403869739ae7f0995e-Paper.pdf

[3] Shanhong Liu. 2020. Global developer population 2024. https://www.statista.com/statistics/627312/worldwide-developer-population/

[4] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. arXiv:1802.04335 [cs.AI]

[5] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[6] David Elliot Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs from Examples. (1975). https://doi.org/10.7916/D89K4K6X

[7] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2012. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (Jan. 2012), 497–518. https://doi.org/10.1007/s10009-012-0223-4

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762

[9] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

[10] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. 2018. NAPS: Natural Program Synthesis Dataset. arXiv:1807.03168 [cs.LG]