AUTOCODER: LEVERAGING TRANSFORMERS FOR AUTOMATIC CODE SYNTHESIS

Mrinal Anand
Indian Institute of Technology,
Gandhinagar, India
mrinal.anand@iitgn.ac.in

Pratik Kayal
Indian Institute of Technology,
Gandhinagar, India
pratik.kayal@iitgn.ac.in

Mayank Singh
Indian Institute of Technology,
Gandhinagar, India
singh.mayank@iitgn.ac.in

Abstract

Synthesizing programs from natural language descriptions is a challenging task. In this paper, we leverage the power of transformer-based language models for the task of program synthesis. We experiment with two variants of transformers and showcase their superior performance than the existing SOTA models. We also discuss the qualitative differences in the learned representation of these two variants. Finally, we compared both these models through the lens of *" degree of memorization"* and demonstrated that the vanilla transformer model has a higher affinity towards memorizing the training data than the other variant.

1 Introduction

Recently, we witnessed a growing interest in the field of Automatic Program Synthesis using program descriptions. Moreover, the field of Natural Language Processing (NLP) is going through a significant paradigm shift towards self-supervised learning, especially with the onset of massive transformer-based models like GPT-3 [1]. Thus, some of the pertinent questions from the current context can be *"Can self-supervised learning paradigm provide near state-of-the-art program synthesis performance?"* We experiment with several variants of Transformer-based encoder-decoder models to showcase that self-supervised models significantly outperform previous neural approaches like RNN and LSTM based attention architectures [2, 3, 4]. We show that the transformer architecture uses relatively lesser training text (only problem description and no I/O pairs) and is able to achieve higher performance than traditional neural code generation models.

Language models based on attention mechanism have demonstrated greater abilities to generate natural language text [5] and have been adapted to several modelling and reasoning tasks [6]. Also, over the past decade, machine learning approaches have been applied to source code text to yield a variety of new tools to support software engineering [7]. Combining these trends raises the question of whether such models designed for natural language understanding and generation can be brought to use in code synthesis. These models provide multiple benefits compared to traditional code generation models. Firstly, it is not necessary for us to encode the grammar for the model, as the model learns this implicitly from the provided training instances. Secondly, these models can be trained on large amount of data, so they can learn about how various code snippets interact with each other and the idiosyncrasies of code descriptions. Finally, these language models allow us to consider a more flexible type of program specification: classically, programs were specified using logical constraints

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

or input-output examples; for modern language models, a program can be specified by a short natural language description.

In this paper, we study how variants of Transformer language models perform when applied to the synthesis of short programs written in general-purpose programming languages like LISP. Our work is especially focused on ALGOLISP dataset. Other popular synthetic datasets include ALGOLISP [4], NAPS [8], Karel [9], WikiSQL [10] and dataset of bash commands [11]. To the best of our knowledge, except for NAPS [8] and Karel [9], no similar code synthesis dataset exists that contains problem description along with the test cases and other meta information. Popular datasets like JAVA [12], WikiSQL [10] only contain problem description and the corresponding code, leading to limitations in evaluating structurally different but logically similar synthesized codes. Although NAPS and Karel contains all the required meta-information, Karel does not deal with any natural language; it is a robotic programming language. On the other hand, in our internal data analysis, NAPS shows several data inconsistencies[1] such as the presence of a long sequence of characters like `abcdabcd`, which conveys no meaning and inconsistent tokenization of sentences.

2 BASELINE AND PROPOSED MODELS

In this section, first we discuss the baseline models present for our work then we discuss our implementation of the neural model AUTOCODER to address the automatic code generation problem.

2.1 Baseline Works

SketchAdapt: SketchAdapt [3] (hereafter *'SA'*) synthesizes programs from textual descriptions as well as input-output test examples. It combines neural networks and symbolic synthesis by learning an intermediate 'sketch' representation. It has been demonstrated empirically that SA recognizes similar descriptive patterns as effectively as pure RNN approaches while matching or exceeding the generalization of symbolic synthesis methods. It is also the state-of-the-art DSL-based code generation model.

SAPS: Structure-Aware Program Synthesis [2] (hereafter, *'SAPS'*) adapts the program synthesis problem under the Neural Machine Translation framework by employing a bi-directional multi-layer LSTM network for generating code sequence corresponding to textual descriptions.

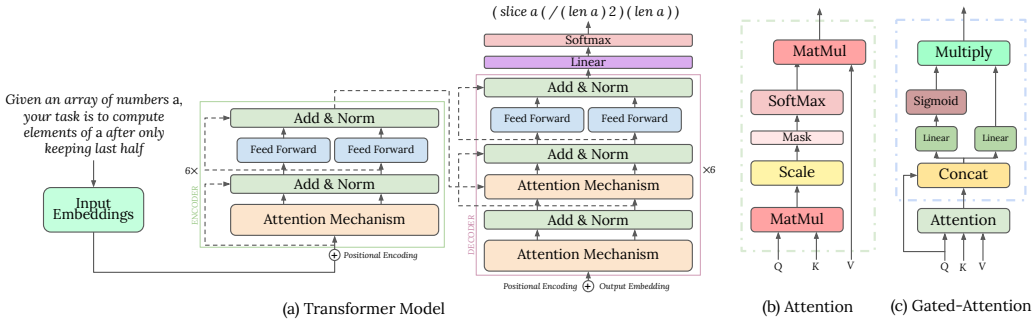
Seq2Tree: The Seq2Tree [4] model consists of a sequence encoder and a tree decoder. The sequence encoder reads the problem description, and a tree decoder augmented with attention computes probabilities of each symbol in a syntax tree node one node at a time.

2.2 AUTOCODER

Recently, Transformers [13] have shown state-of-the-art performance for several Natural Language Processing tasks [14, 15, 16], including machine translation, classification, etc. Inspired by its success, we propose a transformer-based code-generation model to generate code based on natural language descriptions automatically. Specifically, the model encodes the textual description using multiple layers of encoders and then decodes a program token-by-token while attending to the description. The basic pipeline of our proposed model is analogous to the simpler sequence-to-sequence models that are employed for similar generation tasks. These models usually have an encoder-decoder architecture [17, 18]. The encoder maps an input sequence of symbol representations consisting of tokens $x_1, x_2, \cdots, x_n$ to intermediate representation which the decoder then generates an output sequence $y_1, y_2, \cdots, y_m$ of symbols one element at a time. At each step, the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next output symbol. As depicted in Figure 1, we utilize the core Transformer [13] model implementation and propose significant structural alterations in the attention module to develop AUTOCODER.

The Encoding and Decoding Layers: We keep the number of encoder layers (=6) the same as the core Transformer model. Similar to core implementation, we keep the output dimension size as 512 in all the encoding sub-layers of the model as well as in the embedding layers. The decoder side is also stacked with six identical layers. In Figure 1a, the sub-layers of encoder and decoder layers are described using standard notations.

[1]The NAPS dataset is very noisy due to crowd-sourcing.



(a) Transformer Model (b) Attention (c) Gated-Attention

Figure 1: Transformer model. (b) The self-attention mechanism. (c) The gated-attention mechanism.

The different Attention Mechanisms: Attention mechanisms have become an integral part of sequence modeling tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. Specifically, we experiment with two attention mechanisms: (i) vanilla self-cross attention and (ii) gated cross attention. The Vanilla self-cross attention is the basic attention mechanism used in traditional transformer models [13]. The self-attention module relates different positions of an input sequence in order to compute a representation of the input sequence. This module is present in both encoder and decoder layers. The cross attention module relates different positions of an input sequence to the output sequence. This module connects the encoder and decoder components. We term AUTOCODER variant that uses the standard self-cross attention module as **VANILLA-AUTOCODER (VAC)**.

$$f_{SA} = f_{dot}(Q_e, K_e, V_e) = softmax\left(\frac{Q_e K_e^T}{\sqrt{d}}\right) V_e \tag{1}$$

$$f_{CA} = f_{dot}(Q_e, K_d, V_d) = softmax\left(\frac{Q_e K_d^T}{\sqrt{d}}\right) V_d \tag{2}$$

where f_{dot} represents scaled dot product attention [13], (Q_e, K_e, V_e) denotes encoder sequence representations in terms of query, key and value respectively, and (Q_d, K_d, V_d) denotes decoder sequence representation in terms of query, key and value respectively.

The gated cross attention mechanism filters out the unnecessary part of the sequence by attending over the generated cross attention scores f_{CA} and determining the relevant attention. The gated cross attention module (f_{GA}) uses a sigmoidal gating mechanism for filtering out irrelevant cross attention while decoding the output. It generates an *information vector*(i) which carries relevant representation of the input vector and an *attention gate* (g) that filters the relevant attention scores. Now this filtered attention is applied to the information vector to obtain *attended information*, or the relevant information.

$$\begin{aligned} f_{GA} = \sigma(W_q^g Q_e + W_v^g f_{CA} + b^g) \\ \odot(W_q^i Q_e + W_v^i f_{CA} + b^i) \end{aligned} \tag{3}$$

σ denotes sigmoid activation, $\odot$ denotes element-wise product, W_q^i and W_v^i represent weight matrices corresponding to value query and value at *information vector*, respectively, W_q^g and W_v^g represent weight matrices corresponding to value query and value at *attention gate*, respectively. Note that, $\{W_q^i, W_v^i, W_q^g, W_v^g\} \in \mathbb{R}^{d \times d}$ and $\{b^i, b^g\} \in \mathbb{R}^d$. Figure 1c shows the gated cross-attention architecture. We term AUTOCODER variant that uses gated cross attention as **GATED-AUTOCODER (GAC)**.

2.3 AUTOCODER Implementation Details

The AUTOCODER variants are implemented using open-source OpenNMT tool [19]. All experiments run on a four NVIDIA Tesla V100 GPU. During training, we keep six identical layers and eight attention heads for the encoder as well as the decoder. Keeping in mind the smaller vocabulary size of 260, we keep word embedding size equal to 64 dimensions. The model is trained for 300,000 steps with a batch size of 100 for the entire dataset. Adam optimizer [20] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$ is used. With an initial learning rate of 2, we apply Noam's Decay [21] with

$warmup_steps$ equal to 8000. We use ReLU as activation function and a gradient accumulation factor of 2. For regularization, dropout is applied with probability = 0.1, and label smoothing is applied in loss function with $\epsilon_{ls} = 0.1$. The model implementation and evaluation scripts are available at `https://tinyurl.com/4ut7hez5`.

3 Result and Discussion

In this section, we will describe the evaluation metric used to evaluate these code generation models; then we discuss the overall performance and adversarial performance of the transformer and its variant.

3.1 Evaluation Metric

We compute accuracy scores (A) for performance evaluation defined as $A = \frac{n}{N}$, where n is the number of problems for which the generated code passes all the 10 test cases and N is the total number of problems in the holdout test set, this metric helps us to evaluate structurally different but logically similar synthesized codes.

3.2 Quantitative Result

Table 1 compares AUTOCODER variants against baseline systems. Both variants outperformed the previous three baselines on the overall accuracy. Among the two variants, VAC performed marginally better than GAC. To summarize, the results showcase that even simple Transformer variants can result in high gains in code synthesis. However, these scores do not reflect the brittleness of the model.

Model	**Accuracy Scores**
SA	0.958
SAPS*	0.929
SEQ2TREE*	$0.858^{\dagger}$
VAC	**0.968**
GAC	0.963

Table 1: Comparing state-of-the-art code generation models. * represents accuracy scores taken, verbatim, from the corresponding papers due to unavailability of code or pretrained model. † represents accuracy scores on the original test set.

3.3 Qualitative Results

In our analysis, we found that if variables `b` and `d` are interchanged, the model fails to recognize this change and outputs code as if no change has been done on the input sentences. We observe that one of the possible reason for the poor performance of these models is because of the incorrect/poor cross attending. For example, the variable `a` in the output is not attending the corresponding variable `a` in the problem description. We suspect that this incorrect cross attendance adds brittleness to the system and can easily break under adversarial settings. Refer to Appendix A for attention graph.

VAC was able to significantly perform better for problems involving *"finding the length of segment/array"*, *"doing boolean arithmetic operation like greater than, less than, equals to and not equals to"* on the other hand for the problems related to *"central tendency (finding mean or median) of array elements"* or *"finding common subsequence between two arrays"*. GAC was able to significantly outperforms the other variant. Some other classes like *"sorting problems"* and *"problems related to palindrome sequences"* has very high accuracy($> 99\%$) for both the variants. And finally, there are problems related to *"manipulation of array head"* in which both the variants have high error rates. We also observe that, in the case of longer sequences, the error rate is higher for VAC than GAC. However, GAC made frequent mistakes in synthesizing shorter programs.

Although it is evident from Table 1, that overall VAC has higher accuracy than GAC, i.e. more number of correct program has been synthesized in case of VAC. However, we argue that there are significant differences in the learned representation of both variants. On carefully analyzing the

output of both the variants, we observe that more number of structurally different codes(SDC) were synthesized by GAC which passes all the test cases than VAC.

$$SDC = \text{total number of } \textit{`correct programs`} - \text{total number of } \textit{`exact match programs`}$$

where *'correct programs'* are those programs that passes all the test cases while *'exact match programs'* are those that exactly matches the ground truth code. In total, 69 and 52 structurally different codes were synthesized by GAC and VAC. The SDC value has negative correlation with the *"degree of memorization"* of training data. Higher the SDC value, lower is the degree of memorization. The above values indicates that the vanilla transformer model (VAC) tends to memorize the training data that has been feed to it and hence is more susceptible to adversarial perturbations. Moreover, a similar trend was observed in the case of 'almost correct program' (we consider the program to be almost correct when a program passed 50% of the test cases) with 70 and 58 in case of GAC and VAC respectively. Both the variants were able to learn the grammar of ALGOLISP to a very high precision, which is evident from the fact that nearly zero programs were syntactically incorrect or had runtime errors during the inference.

4 Conclusion

In this paper, we experiment with two variants of the transformer model, both variants able to outperform the previous state-of-the-art model on ALGOLISP dataset. We finally demonstrate that the learned representation of both the variants was significantly different from each other and was able to conclude that the vanilla transformer model tends to memorize the training data more as compared to the gated-attention transformer.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[2] Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. Ain't nobody got time for coding: Structure-aware program synthesis from natural language. *arXiv preprint arXiv:1810.09717*, 2018.

[3] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870, 2019.

[4] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[8] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. *arXiv preprint arXiv:1807.03168*, 2018.

[9] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[10] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[11] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[12] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[14] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, 2019.

[15] Naihan Li, Shujie Liu, Yanqing Liu, Sheng Zhao, and Ming Liu. Neural speech synthesis with transformer network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6706–6713, 2019.

[16] Lasha Abzianidze, Rik van Noord, Hessel Haagsma, and Johan Bos. The first shared task on discourse representation structure parsing. In *Proceedings of the IWCS Shared Task on Semantic Parsing*, 2019.

[17] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[18] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[19] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Proc. ACL*, 2017.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604, 2018.

[22] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics.

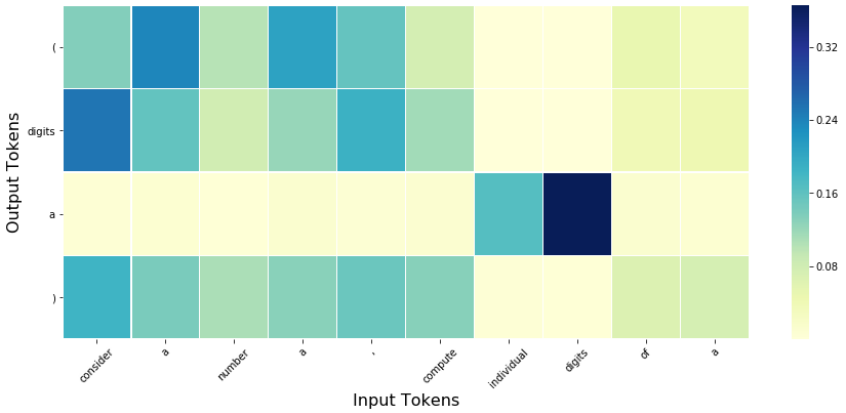


Figure 2: Visualization of Decoder Attention. The variable ‘a’ in the output is not attending the corresponding token ‘a’ in the problem description. We take avg of all attention heads in the fourth layer since it has focused attention while most of the other layers had very broad attention as demonstrated in [22].

A Appendix

Problem Description: consider a number a , compute individual digits of a ?
Code : `( digits ( a ) )`

The high-level idea as to what code to synthesis from the above problem description can be conveyed in just four tokens, “compute digits of a”. We observe that the model does not pay attention to these important words while predicting the output. Figure 2 shows the attention score of each token. It is evident that the variable ‘a‘ in the output is not attending the corresponding token ‘a‘ in the problem description not on other tokens like ’digits’. We suspect that since the variable ‘a‘ is now associated with the token ’digit’, changing the variable name from ‘a‘ to ‘b‘ in the input statement won’t reflect in the synthesized code because of poor cross-attention.