# Are Transformers All That Karel Needs?

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Recent works have shown the incredible promise of using neural networks for the task of program synthesis from input-output examples. In this paper, we propose using Transformer-based architectures as a baseline for the program synthesis task on the Karel dataset. Specifically, we leverage DistillGPT as our decoder model to perform program synthesis for the Karel DSL. We show that changing the model architecture from an LSTM to a transformer based architecture, we are able to significantly improve on supervised learning approaches obtaining a top-1 generalization accuracy of 82.4%. Further, applying execution guided search on the output beams of the model increases the accuracy of our approach to 89.64%.

## 1 Introduction

Programming by example is program generation task such that the generated program satisfies a specification given in the form of input-output pairs. Advances in neural networks have spurred interest in using deep learning techniques to solve problems in the programming languages space.

While domains such as performing arithmetic operations, string manipulation, list sorting, query generation, etc have been studied in literature, generating programs for languages with more complex structures like looping and conditionals is a harder problem to solve.

The Karel programming language [24] was used in the 1980s as an introductory language by Stanford CS students. Its DSL is aimed at controlling a robot in a grid space, where it must navigate to pick and place markers while avoiding the walls. The DSL specification (Fig. 1) has support for conditional and looping constructs, which make it an ideal test bed for evaluating program synthesis approaches.

Devlin et al.[9] introduced the Karel dataset for performing neural program induction. Their task was to learn to represent a program and by training on input-output examples and predicting the output of a program given a new input grid. Our work is focused on program synthesis, whereby we aim to generate a program given the input-output pairs obtained on executing the program.

Bunel et al.[3] introduced using a neural approach for program synthesis using the Karel dataset as a benchmark. They used 5 IO pairs as an input to the model to synthesize the program, while using the 6th pair as a held out test sample on which to evaluate the generalization performance. They presented both supervised and reinforcement learning based approaches and showed that RL naturally achieved better generalization ability since it maximized the probability of generating semantically correct programs.

Since then a number of approaches have been proposed and studied to improve the performance in the Karel task. These approaches include inferring trace executions [29], using execution based decoding [6], learning to execute latent programs [7] and to repair programs after decoding [14].

We posit that transformer based models should be able to achieve much better performance on Karel, even without employing techniques like program repair or modeling intermediate state representations. Transformer based models have been shown to perform well for generating programs from a natural

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

$$
\begin{array}{rcl}
\text{Prog } p & := & \texttt{def run}() \ : \ s \\
\text{Stmt } s & := & \texttt{while}(b) : s \mid \texttt{repeat}(r) : s \mid s_1; s_2 \mid a \\
 & \mid & \texttt{if}(b) : s \mid \texttt{ifelse}(b) : s_1 \ \texttt{else} : s_2 \\
\text{Cond } b & := & \texttt{frontIsClear}() \mid \texttt{leftIsClear}() \mid \texttt{rightIsClear}() \\
 & \mid & \texttt{markersPresent}() \mid \texttt{noMarkersPresent}() \mid \texttt{not } b \\
\text{Action a} & := & \texttt{move}() \mid \texttt{turnRight}() \mid \texttt{turnLeft}() \\
 & \mid & \texttt{pickMarker}() \mid \texttt{putMarker}() \\
\text{Cste } r & := & 0 \mid 1 \mid \cdots \mid 19
\end{array}
$$

Figure 1: Karel DSL specification. Figure from Devlin et al.[9]

language specification as well as to perform code completion in languages like Python. Scholak, Schucher, and Bahdanau[28] show that using vanilla transformers along with some execution guided search are able to perform well in synthesizing SQL queries.

## 2 Related Work

With the advent of neural program synthesis [23, 10], the models are able to work on different domains with little modification and are more robust towards giving generalizable results.

Initial works in neural program synthesis utilized neural networks to prune the search space for traditional program synthesis techniques [16, 22], but then [3] introduced neural models using LSTM layers capable of generating programs. Subsequent works improved upon Bunel et al.'s work while keeping the network architecture relatively unchanged. [29] further showed that using the information from traces of individual examples helps the model in generating better programs. [7, 6, 21] devised techniques to execute the partially generated programs on the inputs to obtain the intermediate states that help the model get a better estimation of the next tokens to be generated. Gupta et al. [14] improve upon execution guided search by introducing a neural debugger to repair the generated programs.

Introduction of Transformers [31] in Natural Language Processing (NLP) has been revolutionary. Transformers overcame pitfalls of LSTMs and GRUs like exploding and vanishing gradients, weakness in modelling long-term dependencies by using a multi-head attention mechanism that can process the input tokens in parallel rather than sequentially. With time, larger and more capable Transformer models were introduced that improved the performance of neural models in different problem statements within NLP [8, 20, 18, 19, 26] and produced comparable performance in computer vision [4, 11, 17] and speech processing as well [5, 32, 33].

Recently, a lot of attention is being paid towards problems such as source code generation, code completion, function naming, bug fixing, etc., in programming languages and impressive results are produced with transformer models. [30] is a general purpose code completion tool for multiple programming languages, [2, 13] used transformers for human language to code generation, [26] introduced an unsupervised neural transcompiler that learns to translate programs between C++, Java and Python even in the absence of parallel training corpus, [12] utilized a BERT based architecture for natural language code search and code documentation generation. ProgRES [1], a recently introduced large scale few-shot program induction benchmark on C++ programs, uses a transformer architecture (BART) as a baseline model for synthesizing programs. This shows that transformers can capture the syntactical information of the programming languages they are trying to model. The results show that large language models are capable to solve programming tasks without the need of adding additional contextual information.

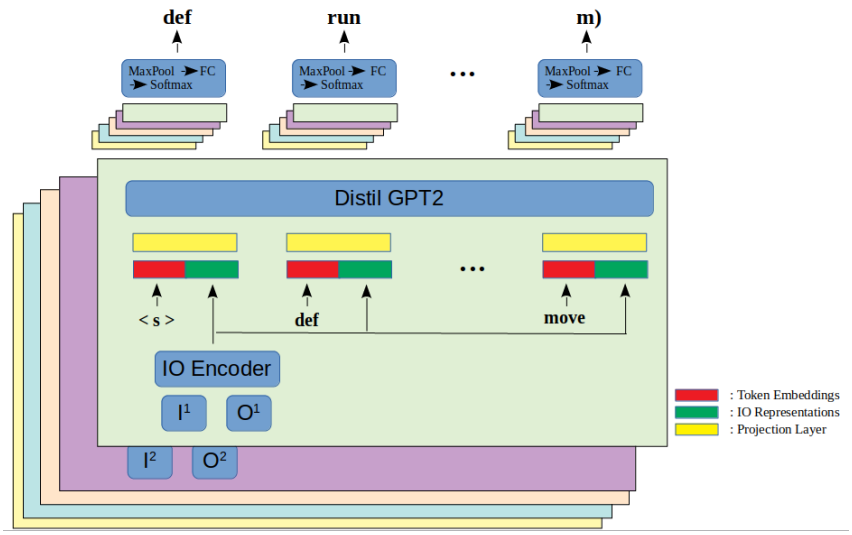


Figure 2: Model Architecture: The architecture is based on [3].

## 3 Approach

Our approach is inspired by the promise shown by transformer architectures in the NLP and program synthesis community. Our input-output encoding is handled by a convolutional neural network as described in [3]. For our decoder, we used DistillGPT2, available on [15], as our base transformer architecture. It uses the technique described in [27] to distill the GPT-2 model down to get a model which has 37% lesser weights compared to the original architecture. We chose this model as it requires a lower computational cost to train the model. Both the IO encoder and DistillGPT decoder layers are initialized with random weights and both are learned end-to-end. Our model architecture is shown in Figure 2.

During inference, we take the top beam output of the model as the chosen program for evaluation. When adding simple execution guidance (SEG), we sample the top 50 beams of the model, and use the Karel interpreter to evaluate them on the 5 IO pairs. We take the top ranked program which satisfies all the 5 pairs, before using this to evaluate against the 6th held out IO sample. If no beam satisfies all the 5 pairs, then we return a NULL program which counts as an incorrect output during evaluation. This method of using the interpreter as guidance for beam selection is more rudimentary than methods detailed in the past [6] and is similar to the greedy search employed by [14]. However, since we do not execute partial programs, we do not need to add measures to deal with issues surrounding sampling for looping and branching constructs.

## 4 Experimental Setup

The Karel dataset consists of a train/validation/test split of around 1.1 million synthetically generated programs written in the Karel language for the training set, and 2.5k programs each for the validation and testing set. Each program is accompanied by 6 input output(IO) pairs. These IO pairs are image representations of the state of the Karel robot in the grid world. We use the programs and IO pairs in the training set to train our model, and report results by running our inference procedure on the IO samples in the test set.

Following [3], we set up the task as supervised sequence generation conditioned on a set of input-output representations. The decoder is fed a combined representation of the IO encoding and the output of the previous timestep. We used Adam optimizer, batch size 8 (due to computational constraints), with initial learning rate of 1e-4. We used a linear learning rate scheduler with warmup of 139606 steps (around 1 epoch) and gradient clipping to ensure that the transformer model doesn't diverge during training [25] and trained for 20 epochs. The model was trained on a DGX-1 machine with an Intel Xeon E5-2698 CPU, 32GB of available system memory to the training job, and an Nvidia Tesla V100 GPU, using an allocation of 16GB VRAM.

| Method | Top-1 Gen | Exact Match |
|---|---|---|
| Bunel et al. [3] | 73.67% | 39.94% |
| Shin, Polosukhin, and Song [29] | 81.30% | 42.80% |
| Chen, Song, and Tian [7] | 83.68% | 41.12% |
| Chen, Liu, and Song [6] | 86.04% | 39.40% |
| Gupta et al. [14] | **89.80%** | 43.48% |
| Transformer Decoder | 82.40% | 43.36% |
| Transformer + SEG | 89.64% | **44.80%** |

Table 1: Results on Karel test dataset.

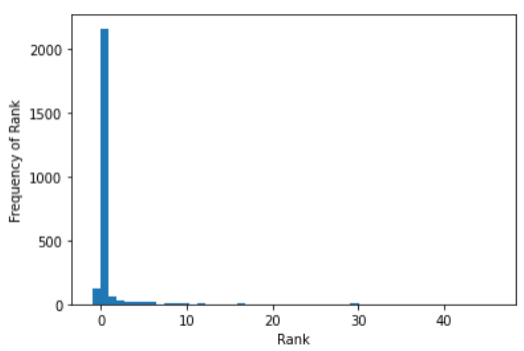


Figure 3: Distribution of beams chosen by execution guidance. x-axis shows the rank of the beams, y-axis shows the number of samples chosen by SEG for that rank out of 2.5k test samples. Rank = -1 indicates that the SEG model was unable to find a program from the 50 beams that satisfied all 5 IO samples.

## 5 Results

Table 1 shows our top-1 generalisation and exact match accuracies on the Karel test dataset. Our generalisation accuracy is calculated by using the program obtained by feeding 5 input-output pairs to the model and evaluating this against the 6th held out IO sample. For Transformer + Simple Execution Guidance, this accuracy is calculated on the program selected after searching the beams. We obtain a generalisation accuracy of 82.4% on our base transformer model which increases to 89.64% after adding the execution guidance. We also obtain an exact match accuracy of 44.8% after performing SEG during inference.

While analyzing the results (Fig. 3) on the test set of 2500 programs, we found that in 86.4% (or 2160 instances), the execution guidance was selecting the top ranked beam from the model output. In 124 cases, our inference method returned a NULL program, indicating that no beam satisfied all 5 input-output pairs. The remaining samples tailed off as per the ranking given by the model.

## 6 Conclusion

We report results for program synthesis on Karel by using transformers as the base architecture, replacing the LSTM-based models that have been reported earlier. We show that this change in architecture results in an improvement in generalisation accuracy over the base model without having to model intermediate trace representations. Additionally, even performing simple execution guided search over the beam outputs, we are able to achieve near state-of-the-art results on the task. We feel that steps such as learning trace representations or program repair are complimentary to our work and could be used to boost performance in conjunction with transformer-based models.

## References

[1] Ferran Alet et al. “A large-scale benchmark for few-shot program induction and synthesis”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 175–186.

[2] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[3] Rudy Bunel et al. "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis". In: *International Conference on Learning Representations*. 2018.

[4] Nicolas Carion et al. "End-to-end object detection with transformers". In: *European Conference on Computer Vision*. Springer. 2020, pp. 213–229.

[5] Feng-Ju Chang et al. "End-to-End Multi-Channel Transformer for Speech Recognition". In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 5884–5888.

[6] Xinyun Chen, Chang Liu, and Dawn Song. "Execution-guided neural program synthesis". In: *International Conference on Learning Representations*. 2018.

[7] Xinyun Chen, Dawn Song, and Yuandong Tian. "Latent Execution for Neural Program Synthesis Beyond Domain-Specific Languages". In: *arXiv preprint arXiv:2107.00101* (2021).

[8] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[9] Jacob Devlin et al. "Neural program meta-induction". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 2077–2085.

[10] Jacob Devlin et al. "Robustfill: Neural program learning under noisy i/o". In: *International conference on machine learning*. PMLR. 2017, pp. 990–998.

[11] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021.

[12] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 2020, pp. 1536–1547.

[13] Leo Gao et al. "The Pile: An 800GB Dataset of Diverse Text for Language Modeling". In: *arXiv preprint arXiv:2101.00027* (2020).

[14] Kavi Gupta et al. "Synthesize, execute and debug: Learning to repair for neural program synthesis". In: *Proceedings of the 34th Conference of Neural Information Processing Systems (NeurIPS)*. 2020.

[15] Huggingface. *DistillGPT model checkpoint*. 2021. URL: https://transformer.huggingface.co/model/distil-gpt2.

[16] Ashwin Kalyan et al. "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *International Conference on Learning Representations*. 2018.

[17] Salman Khan et al. "Transformers in vision: A survey". In: *arXiv preprint arXiv:2101.01169* (2021).

[18] Zhenzhong Lan et al. "Albert: A lite bert for self-supervised learning of language representations". In: *arXiv preprint arXiv:1909.11942* (2019).

[19] Mike Lewis et al. "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension". In: *arXiv preprint arXiv:1910.13461* (2019).

[20] Yinhan Liu et al. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[21] Maxwell Nye et al. "Representing Partial Programs with Blended Abstract Semantics". In: *International Conference on Learning Representations*. 2020.

[22] Augustus Odena et al. "{BUSTLE}: Bottom-Up Program Synthesis Through Learning-Guided Exploration". In: *International Conference on Learning Representations*. 2021.

[23] Emilio Parisotto et al. "Neuro-symbolic program synthesis". In: *arXiv preprint arXiv:1611.01855* (2016).

[24] Richard Pattis, J Roberts, and M Stehlik. "Karel the robot". In: *A gentele introduction to the Art of Programming* (1981).

[25] Martin Popel and Ondřej Bojar. "Training Tips for the Transformer Model". In: *The Prague Bulletin of Mathematical Linguistics* 110.1 (2018), pp. 43–70.

[26] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[27] Victor Sanh et al. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS*. 2019.

[28] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. “PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models”. In: *Empirical Methods in Natural Language Processing*. 2021.

[29] Eui Chul Shin, Illia Polosukhin, and Dawn Song. “Improving neural program synthesis with inferred execution traces”. In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 8917–8926.

[30] Alexey Svyatkovskiy et al. “Intellicode compose: Code generation using transformer”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1433–1443.

[31] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[32] Changhan Wang et al. “Fairseq S2T: Fast Speech-to-Text Modeling with Fairseq”. In: *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing: System Demonstrations*. 2020, pp. 33–39.

[33] Yongqiang Wang et al. “Transformer-based acoustic modeling for hybrid speech recognition”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 6874–6878.