Are Transformers All That Karel Needs?

Abhay Garg Anand Sriraman Kunal Pagarey Shirish Karande
TCS Research, Pune, India
{abhay.garg1, anand.sriraman, kunal.pagarey, shirish.karande}@tcs.com

Abstract

Recent works have shown the promise of using neural networks for the task of program synthesis from input-output examples. The Karel dataset has been a benchmark for evaluating program synthesis approaches. Several techniques have been proposed to use neural guided program synthesis with Karel being used as a baseline. Most of these techniques use an LSTM based model for decoding and improve performance by proposing complex algorithmic additions, such as using inferred execution traces, latent execution of partial programs and debugging generated programs. We observe that by changing the base architecture to a transformer based one, specifically GPT2, we are able to apply simple execution guidance on top to achieve a generalization accurary of 89.64%, which is within 2.36 percentage points of the current state-of-the-art on Karel which uses ensembling.

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 [28] 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.[10] 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 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 to evaluate the generalization performance.

Since then a number of approaches have been proposed and studied to improve the performance in the Karel program synthesis task. These include inferring trace executions [32], using execution based decoding [6], learning to execute latent programs [7] and to repair programs after decoding [16]. These methods focus on making complex changes to how the input and output to the model is processed, while keeping the base architecture as an LSTM. We posit that transformer based models should be able to achieve much better base performance on Karel which can be improved further by adding relatively simpler execution guided search.

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

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

Figure 1: Karel DSL specification. Figure from Devlin et al.[10]

2 Related Work

With the advent of neural program synthesis [27, 11], 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 [19, 25], 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. [32] showed that using the information from traces of individual examples helps the model in generating better programs. [7, 6, 24] 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. While [6] uses methods to sample executable partial programs from the output beams to perform search, [7] try to execute partial programs in the latent space, removing the need for algorithmic methods of sampling partial programs. Gupta et al. [16] improve upon execution guided search by introducing a neural debugger to repair the generated programs.

Introduction of Transformers [34] 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 [9, 23, 21, 22, 30] and produced comparable performance in computer vision [4, 12, 20] and speech processing as well [5, 35, 36].

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. [33] is a general purpose code completion tool for multiple programming languages, [2, 14, 26, 8, 17] used transformers for natural language to code generation, [30] introduced an unsupervised neural transcompiler that learns to translate programs between C++, Java and Python even in the absence of parallel training corpus, [13] utilized a BERT based architecture for natural language code search and code documentation generation. Scholak, Schucher, and Bahdanau[31] show that using vanilla transformers along with some execution guided search are able to perform well in synthesizing SQL queries. 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 of solving programming tasks without the need of adding additional contextual information.

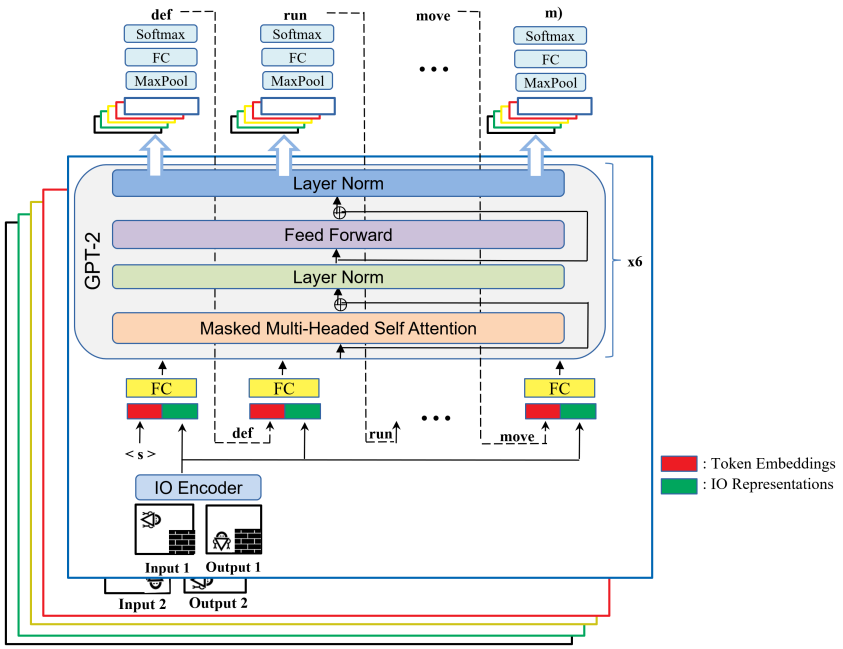

Figure 2: Model Architecture: The architecture is based on [3].

3 Approach

3.1 Dataset

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.

3.2 Model Architecture

Our input-output encoding is handled by a convolutional neural network as described in [3]. For our transformer decoder, we used GPT2 architecture with 6 layers as our base model. We downloaded the model available on [18] as our starting point for the transformer, but initialized both the IO encoder and GPT2 decoder layers with random weights which are then learned end-to-end. Our model architecture is shown in Figure 2.

3.3 Training Hyperparameters

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 [29] 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
Leveraging Grammar + LSTM [3]	73.67%	39.94%
IO->Trace->Program [32]	81.30%	42.80%
Latent Execution [7]	83.68%	41.12%
Execution Guided Decoding [6]	86.04%	40.88%
Learning to Repair [16]	89.80%	43.48%
Execution Guided Decoding (Ensemble) [6]	92.00%	47.08%
LSTM Decoder	73.48%	40.88%
LSTM + SEG	84.48%	43.28%
Transformer Decoder	82.40%	43.36%
Transformer + SEG	89.64%	44.80%
Transformer + Debugger	**90.44%**	**44.88%**

Table 1: Results on Karel test dataset.

3.4 Simple Execution Guidance for Search

During inference, we take the top ranked 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 [16]. 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 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. We compare our results against previous approaches published for program synthesis. Chen, Liu, and Song [6] report results for both a single decoder model, as well as using ensemble approaches. We have reported both results here, however our approach only uses a single model during inference. For SEG results, this accuracy is calculated on the program selected after searching the beams. We observe a generalisation accuracy of 82.4% on our base transformer model which increases to 89.64% after adding the execution guidance. We also observe an exact match accuracy of 44.8% after performing SEG during inference.

We also conducted experiments to validate the advantage of using a transformer model on the current state-of-the-art which is the SED approach by Gupta et al. We fed the output beams of our transformer model to the debugger LSTM trained without fine-tuning as per [16]. We used an implementation available on GitHub [15]. With this approach, mentioned as Transformer + Debugger in Table 1, we achieved a top-1 generalisation accuracy of 90.44%. One thing to note here, we did not perform the fine-tuning step which was described by Gupta et al. due to memory constraints in our system.

4.1 Error Analysis

While analyzing the results 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 SEG inference method returned a NULL program, indicating that no beam satisfied all 5 input-output pairs. Figure 3a shows that accuracy of the model decreases as the length of the ground truth program increases. Figure 3b shows that in the majority of cases, the programs generated by the model are shorter than the ground truth program. This bias could perhaps be rectified by balancing the length of the programs in the dataset. Figure 3c shows that SEG seems to choose longer programs when the top beam doesn't satisfy all 6 IO pairs. Figure 3d analyzes the programs that satisfied all 6 IO pairs, while not being an exact match to the ground truth. It shows the number of programs that have the same trace as the target program for N IO pairs. This shows that the gap between the

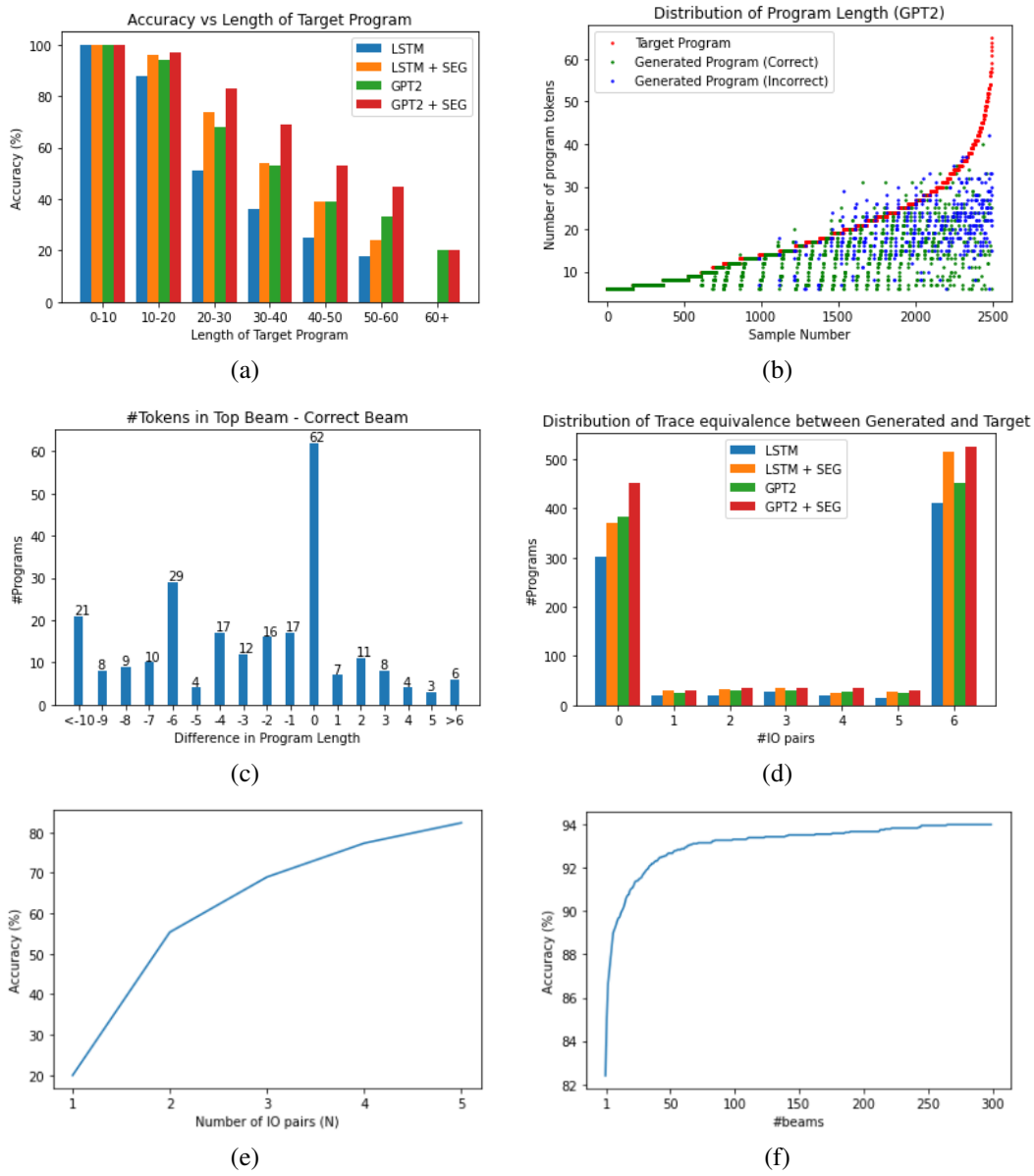

Figure 3: Analysis of the model output.

generalization and exact match accuracies could be explained by the fact that two programs could have the same trace for the IO pairs despite being not an exact match. Figure 3e shows that the accuracy of the model increases as more IO pairs are provided to the model. Figure 3f shows that the accuracy of the model shows a rising trend as the number of beams searched during SEG is increased.

5 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 a significant improvement in generalisation accuracy. Further, adding even performing simple execution guided search over the beam outputs, we are able to achieve performance at par with more complex approaches.

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: *Thirty-Fifth Conference on Neural Information Processing Systems*. 2021. URL: https://openreview.net/forum?id=_nRSyha2SP.

[8] Xinyun Chen et al. "Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2021, pp. 2169–2181.

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

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

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

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

[13] 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.

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

[15] Kavi Gupta. *SED Implementation Reference*. 2020. URL: https://github.com/sunblaze-ucb/SED.

[16] 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.

[17] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: https://openreview.net/forum?id=sD93GOzH3i5.

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

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

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

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

[22] 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).

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

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

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

[26] Augustus Odena et al. “Program Synthesis with Large Language Models”. In: *n/a*. n/a. n/a, 2021, n/a.

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

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

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

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.