Improving Neural Program Synthesis with Inferred Execution Traces

Richard Shin [1 *] **Illia Polosukhin** [2] **Dawn Song** [1]

1. Introduction

Several recent papers have proposed neural network-based approaches to *program synthesis* from input/output examples (Parisotto et al., 2017; Devlin et al., 2017; Bunel et al., 2018). These methods use an end-to-end encoder-decoder approach, where a neural network learns to generate a program from an encoding of a program specification (a set of input/output examples) from a large synthetic training dataset.

Given that an execution trace is a strict superset of an input/output example, intuition suggests that program synthesis from traces should be easier than synthesis from I/O. Indeed, work such as Reed & de Freitas (2016) have successfully used traces in program induction. However, execution traces are more challenging for the end user to specify, so it is hard to reap their benefits. In this work, we use the insight that if encoder-decoder neural networks can synthesize programs from input/output examples, they should also be able to infer execution traces. Thus, we can split the problem into two steps: use input/output examples to infer execution traces, and then use execution traces to infer the program. Our empirical results show that this simple modification leads to state-of-the-art results on the Karel (Pattis, 1981) program synthesis task, improving upon (Bunel et al., 2018) from 77.12% to 81.3% accuracy.

Our analysis shows greater accuracy on programs of varying lengths and complexities, demonstrating the general utility of the approach. This is despite the fact that we only use straightforward maximum likelihood training, which is easier to tune than reinforcement learning methods of prior work.

2. Background

Karel is an educational programming language (Pattis (1981)). It features an agent inside a grid world, where certain cells can contain *markers* or *walls*. The agent starts at some cell in the grid (which may contain markers but no obstacle), and has `move`, `turn{Left,Right}` as actions to move, and `{pick,put}Marker` to manipulate markers. The language contains `if`, `ifElse`, `while` constructs with conditionals `{front,left,right}IsClear`, `markersPresent`, and their negations. `repeat` allows for a fixed number of repetitions.

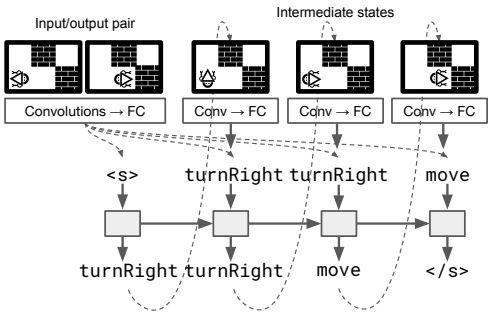

Figure 1. Architecture of I/O → TRACE model.

Our work is based on Bunel et al. (2018), which applied a neural encoder-decoder approach to Karel program synthesis, similar to the work of Devlin et al. (2017) and Parisotto et al. (2017) which was for a string-editing domain. Bunel et al. (2018) used both supervised learning with a randomly-generated synthetic dataset to train their model, as well as a reinforcement learning-based approach to further improve the model's program synthesis accuracy. As part of their work, they have developed a deep learning architecture for Karel program synthesis which we use as the basis for our approach.

3. Approach

3.1. Predicting execution traces from input/output pairs

In this work, an execution trace refers to an ordered set of actions: $(action_1, ..., action_M)$. In the case of Karel, actions are `move`, `turn{Right, Left}`, `{put, pick}Marker`. For a given original training example $(\pi, \{(I_1, O_1), ..., (I_N, O_N)\})$, we can generate N training examples $(I_1, O_1, (action_1, \cdots, action_M)_1), \cdots$, for trace

[*]Work partially performed at NEAR. [1]Computer Science Division, University of California, Berkeley, Berkeley, California, USA [2]NEAR, San Francisco, California, USA. Correspondence to: Richard Shin <ricshin@cs.berkeley.edu>.

Published at the ICML workshop Neural Abstract Machines & Program Induction v2 (NAMPI) — Extended Abstract, Stockholm, Sweden, 2018.

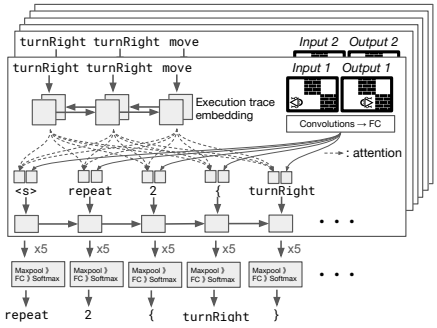

Figure 2. Architecture of TRACE → CODE model. We follow the architecture in Bunel et al. (2018), but add an execution trace input.

prediction by running π on these I/O pairs and recording the actions taken by the program.

Figure 1 shows the deep learning model architecture we used for predictiong execution traces. To encode the input/output examples, we use a convolutional neural network with a final fully-connected layer, taken from (Bunel et al., 2018). To generate the sequence of actions, we use a two-layer LSTM decoder.

3.2. Synthesizing programs from input/output examples and execution traces

To create a model which uses both a set of input/output examples and execution traces for generating the desired program, we started with the architecture from Bunel et al. (2018) and extend it to also take the execution trace as an input. Specifically, we add a bidirectional LSTM responsible for producing an embedding of the execution trace for each step in the trace. See Figure 2 for a visual depiction of the overall model.

To obtain the execution traces for training this model, we used the I/O → TRACE model from Section 3.1 to infer a valid trace for the given I/O pair in the training data. This trace may deviate from the actions taken by the true program even if the final state is identical, as certain sequences of actions can be omitted or permuted without any effect on the final state. Nevertheless, we will only have access to the inferred trace at inference time, so it is useful to match the training and test distributions more closely.

4. Experiments

To train and test our models, we used the same dataset as (Bunel et al., 2018), from `https://bit.ly/karel-dataset`. Each entry in the dataset contains a Karel program and 6 input/output pairs which satisfy that program; we show 5 pairs to the model and hold out the last.

When evaluating the model, we use beam search with beam size 50. We report **Top-K Exact Match**, which measures how often one of the top K output programs of the model textually matches the original program exactly; **Top-K Generalization**, which denotes the fraction of test instances for which one of the top K output programs will have the correct behavior across the 5 input/output examples used as specification, as well as the held-out 6th example; **Top-K Guided Search**, where we consider the top K program outputs in order, from most likely to least likely, test each candidate program on the 5 input/output examples that specify the program, and then test the first one correct on all 5 on the held-out 6th example.

Table 1. Comparison of our best model with previous work from (Bunel et al., 2018).

	Top-1		Top-50	
	Exact	**Gen.**	**Guided**	**Gen.**
MLE (Bunel et al., 2018)	39.94%	71.91%	–	86.37%
RL_beam_div_opt (Bunel et al., 2018)	32.17%	77.12%	–	85.38%
I/O → CODE, MLE	40.1%	73.5%	84.6%	85.8%
I/O → TRACE → CODE, MLE	**42.8%**	**81.3%**	**88.8%**	**90.8%**

Table 2. Comparing performance on different slices of data.

Slice	% of dataset	I/O → CODE	I/O → TRACE → CODE	Δ%
No control flow	26.4%	100.0%	100.0%	+0.0%
Only Conditions	15.6%	87.4%	91.0%	+3.6%
Only Loops	29.9%	91.3%	94.3%	+3.0%
With some control flow	73.6%	79.0%	84.8%	+5.8%
Program length 0-15	44.8%	99.5%	99.5%	+0.0%
Program length 15-30	40.7%	80.8%	86.9%	+6.1%
Program length 30+	14.5%	48.6%	61.0%	+12.4%

In Table 1, we compare our best I/O → TRACE → CODE model (created by gluing together I/O → TRACE and TRACE → CODE) against the previous work of Bunel et al. (2018). We reimplemented their MLE model (labeled as I/O → CODE), obtaining slightly better results compared to theirs. We note that we did not implement the RL_beam_div_opt training method of Bunel et al. (2018), and so our results are all based on MLE training. Nevertheless, our I/O → TRACE → CODE method outperforms all others on all metrics, including the best result in Bunel et al. (2018).

We also analyzed how models performed on various slices of the test data in Table 2: programs with no control flow (only actions); programs with conditionals (`if` or `ifElse`) but not loops (`repeat` or `while`); programs with loops but no conditionals; and programs containing at least one control flow element. We also partitioned the data depending on the length of the gold program into three buckets. We can observe that I/O → TRACE → CODE improves upon I/O → CODE within every slice of the data. The magnitude of the improvement is most significant on long programs.

References

Bunel, Rudy, Hausknecht, Matthew, Devlin, Jacob, Singh, Rishabh, and Kohli, Pushmeet. Leveraging grammar and reinforcement learning for neural program synthesis. *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Xw62kRZ.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pp. 990–998, 2017.

Parisotto, Emilio, Mohamed, Abdel-rahman, Singh, Rishabh, Li, Lihong, Zhou, Dengyong, and Kohli, Pushmeet. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

Pattis, Richard E. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016. URL http://arxiv.org/abs/1511.06279.