Synthetic Datasets for Neural Program Synthesis

Richard Shin [1] **Neel Kant** [1 2] **Kavi Gupta** [1] **Christopher Bender** [1 2] **Brandon Trabucco** [1 2] **Rishabh Singh** [3] **Dawn Song** [1]

1. Introduction

Neural program synthesis approaches learn techniques to search programs in a domain-specific language (DSL) trained on a large corpus of DSL programs. This technique has been instantiated for many domains including learning string transformations in RobustFill (Devlin et al., 2017b; Parisotto et al., 2017), array programs in DeepCoder (Balog et al., 2016), and more recently in Karel programs (Devlin et al., 2017a; Bunel et al., 2018). These approaches typically train the models on a large number of synthetically generated programs and the corresponding synthetic specifications, relying on the hypothesis that if the model is able to accurately predict arbitrary programs in the DSL, then it can also predict well on new test distributions.

In this paper, we consider the recently proposed synthesis model for Karel (Bunel et al., 2018). We find that on many distributions of input examples and DSL programs, the Karel synthesis model trained on existing synthetic data performs poorly. In particular, choosing different I/O examples for the same test programs or choosing different programs significantly decreases the prediction accuracy, in some cases leading to even 0%.

We propose a new methodology for generating different distributions over programs in the DSL, as well as over inputs which specify the program. Consider a DSL D specified using a context-free grammar that denotes its syntax, and let the valid input space I be specified using another other enumerable space. Formally, we define two sets of *salient* random variables $X = \{X_1, \cdots\}$ and $Z = \{Z_1, \cdots\}$, where the random variable X_i denotes a certain important feature in the DSL D and the random variable Z_j denotes a feature for the input space.

Karel programs control an agent in a grid world; each cell in the grid can contain markers, a wall, or neither. We devised the following salient random variables to describe the input space of grid worlds: *grid size*; *marker ratio*, fraction of cells with at least one marker; *wall ratio*, fraction of cells which contain a wall; *marker count*, random variable which specifies how many markers are in a cell which has markers. For the program space, we used: *program size*, size of the program in terms of number of tokens; *control flow ratio*, number of control flow structures appearing in the program; *nested control flow*, the amount of control flow nesting in programs (e.g. while inside if); *odd/even length*, programs of even and odd lengths.

[1]University of California, Berkeley [2]Machine Learning at Berkeley [3]Google Brain. Correspondence to: Richard Shin <richshin@cs.berkeley.edu>.

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

2. How the Existing Model Fails

By imposing various distributions over the salient random variables when producing the input/output specifications and target programs which are the test set, we observe much lower accuracies of the previous Karel synthesis models (Bunel et al., 2018) compared to the original test set.

Input space: uniform distribution. We first generated grids such that they would follow a distribution that is as uniform as possible in the salient features. Specifically, we sampled the grid size, marker ratio, wall ratio, and marker count uniformly. On this dataset, the model achieved accuracy of only 27.9%, which was a drop of 44.6pp from the existing test set's accuracy of 73.52%.

Input space: narrow distributions. We further investigated the drop in performance noted above by synthesizing "narrower" datasets that captured different parts of the joint probability space over the salient input random variables. For each narrow dataset, we selected r_{wall} and r_{marker} (both between 0 and 1) as well as a distribution $\mathcal{D}_{\text{marker count}}$ which would be the same for all I/O grids. In our experiments, we primarily used 3 different distributions for $\mathcal{D}_{\text{marker count}}$: $Geom(0.5)$ truncated at 9, $\mathcal{U}\{1, 9\}$, and $10 - Geom(0.5)$ (10 minus a sample from $Geom(0.5)$) truncated at 1. The top row of Table 1 shows the baseline model's performance on these test sets.

Program space: actions only. Intuitively, much of the difficulty in the Karel program synthesis task should come from inferring the control flow statements, i.e. `if`, `ifElse`, and `while`. Synthesizing a Karel program that only con-

Table 1. Generalization accuracies of baseline model and model trained on uniformly distributed salient random variables on selected datasets. G, U, and A stand for $Geom(0.5), \mathcal{U}\{1, 9\}$ and $10 - Geom(0.5)$ respectively.

r_{wall}	0.05			0.25			0.65			0.85		
r_{marker}	0.85			0.65			0.25			0.05		
$\mathcal{D}_{\text{marker count}}$	G	U	A	G	U	A	G	U	A	G	U	A
Baseline (%)	24.30	1.32	0.04	21.08	2.98	0.08	16.63	13.31	6.63	15.99	12.88	12.98
Uniform (%)	63.90	65.68	65.75	59.96	60.32	59.13	62.0	62.94	63.92	73.83	75.66	76.25
Δ	+39.6	+64.36	+65.71	+38.88	+57.34	+59.05	+45.37	+49.63	+57.29	+57.84	+62.78	+63.27

Table 2. Results on programs only containing actions. The accuracy on action-only programs in the existing test set is 99.24%.

	Program length							
Model type	1	2	3	4	5	6	7	8
Baseline	16.00%	30.00%	44.24%	52.88%	56.56%	66.94%	67.16%	73.06%
Action-Only Augmented	20.00%	41.60%	52.24%	61.72%	63.04%	72.20%	72.74%	78.12%

Table 3. Partitioning training and test sets based on whether the number of tokens in the program is odd or even. Testing on the other configuration leads to a significant decline in performance.

Train	Test	Exact match	Generalization
Odd	Odd	36.03%	59.26%
Odd	Even	0.84% (-35.19 p.p.)	41.69% (-17.57 p.p.)
Even	Even	40.78%	62.58%
Even	Odd	0.84% (-39.94 p.p.)	40.74% (-21.84 p.p.)

tains actions is intrinsically a much more straightforward task. We performed an experiment using test datasets generated by enumerating action-only programs of various lengths. The top row of Table 2 shows the results.

Program space: odd/even length. In this experiment, we introduced a divergence in the distributions of training and test programs by partitioning them based on whether they contain an odd or even number of tokens. After we split the training dataset into these two parts, we trained a separate model for each part. We then tested each model on each of the two parts of the test dataset, for a total of four evaluations. Table 3 shows the accuracy obtained in each setting. As the training dataset is smaller, the model overall does not perform as well. Nevertheless, accuracy drops significantly when we test a model on the other partition.

Program space: complex DSL constructs. We examined whether the model could synthesize programs which require nested conditional constructs; these programs were relatively rare in the training dataset. We generated an evaluation dataset comprised solely of programs that contained `while` inside `while` statements, and another dataset in which all programs had `while` inside `if` statements. We found that the model fared very poorly on these datasets, achieving only 0.64% and 2.23% accuracy respectively.

3. Training the Model on New Datasets

Various imbalances of the salient random variables in the existing training data could have caused the gaps in performance seen in the previous section. Thus, a natural solution is to to train using datasets constructed to avoid undesirable skews in the salient random variables.

Training datasets with uniform I/O. We generated a training dataset by taking the programs of the existing training set and synthesizing I/O pairs with the approach from *Input space: uniform distribution*. We trained a model on this data and then evaluated it on the same set of narrow distribution evaluation datasets as mentioned in *Input space: narrow distributions*. Table 1 compares how this new model performs to the baseline model. The model trained on uniform I/O distributions maintains much higher inference accuracy on the narrow input space distribution test sets than the baseline model. Note that the uniform I/O distribution is not simply a union of the narrow distributions, but nevertheless intended to cover all possible specifications.

Adding action-only programs to training data. We observed that the model fails to do well on action-only programs despite their relative simplicity. We suspected that this was due to non-uniformity in the distribution of program length. A principled way to counteract this would be to introduce uniformity into the distribution of program length as it is a salient output random variable. We proceeded by adding 20,000 action-only programs of each length, 1 through 20, to the given training set in order to train a new model. Table 2 shows the clear improvement in performance in this metric by the newly trained model over the baseline.

References

Balog, Matej, Gaunt, Alexander L., Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bunel, Rudy, Hausknecht, Matthew, Devlin, Jacob, Singh, Rishabh, and Kohli, Pushmeet. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

Devlin, Jacob, Bunel, Rudy R., Singh, Rishabh, Hausknecht, Matthew J., and Kohli, Pushmeet. Neural program meta-induction. In *NIPS*, pp. 2077–2085, 2017a.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy I/O. In *ICML*, pp. 990–998, 2017b.

Parisotto, Emilio, Mohamed, Abdel-rahman, Singh, Rishabh, Li, Lihong, Zhou, Dengyong, and Kohli, Pushmeet. Neuro-symbolic program synthesis. *ICLR*, 2017.