SYNTHETIC DATASETS FOR NEURAL PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

The goal of program synthesis is to automatically generate programs in a particular language from corresponding specifications, e.g. input-output behavior. Many current approaches achieve impressive results after training on randomly generated I/O examples in limited domain-specific languages (DSLs), as with string transformations in RobustFill. However, we empirically discover that applying test input generation techniques for languages with control flow and rich input space causes deep networks to generalize poorly to certain data distributions; to correct this, we propose a new methodology for controlling and evaluating the bias of synthetic data distributions over both programs and specifications. We demonstrate, using the Karel DSL and a small Calculator DSL, that training deep networks on these distributions leads to improved cross-distribution generalization performance.

1 INTRODUCTION

Program synthesis is one of the central problems in Artificial Intelligence studied from the early days (Manna & Waldinger, 1971; Waldinger & Lee, 1969) and has seen a lot of recent interest in the machine learning and programming languages community (Alur et al., 2013; Gulwani et al., 2012; 2015; Muggleton, 1991; Lin et al., 2014; Solar-Lezama, 2013). The recent neural approaches can be broadly classified into two categories: *program induction* and *program synthesis*. Both approaches share the objective of learning program semantics but do so in different ways. Program induction aims to embed the semantics of a particular algorithm into a differentiable model trained end-to-end, whereas the goal of program synthesis is for a model to learn the semantics of a domain-specific language (DSL) and produce programs defined by corresponding specifications. Both problems necessitate large datasets, either of I/O pairs in the case of program induction, or programs with corresponding I/O pairs in the case of program synthesis.

However, since large datasets for program induction and synthesis tasks do not exist, these approaches train models on large synthetically generated datasets. Presumably, if a model can accurately predict *arbitrary* program outputs (for induction) or programs in the DSL (for synthesis) then it has likely learnt the correct algorithm or DSL semantics.

Although this approach has led to some impressive synthesis results in many domains, synthetically generating datasets that cover all DSL programs and the corresponding input space can be problematic, especially for more complex DSLs like *Karel the Robot* (Pattis, 1981) which includes complex control-flow primitives (while loops and if conditionals) and operators. Likewise, for induction tasks, the sampling procedure for program specifications may lead to undesirable biases in the training distribution that inhibit strong generalization.

In this paper, we consider two problem settings. The first is the Karel domain and the recently proposed Karel synthesis model (Bunel et al., 2018). We identify many distributions of input examples and DSL programs for which the Karel synthesis model performs poorly. The second problem setting is a program induction problem in which a model is trained to execute and predict the output of simple arithmetic expressions, which we denote the *Calculator* domain. We considered common synthetic data generation strategies including one from `tensor2tensor` (Vaswani et al., 2018), an open-source deep learning library. Upon analysis, we find evidence of undesirable artifacts resulting from certain biases in the generation algorithm.

Our results indicate that models trained with common methodologies for synthesizing datasets fail to learn the full semantics of the DSL, even when they perform well on a test set, and suggest the need of a more principled way to generate synthetic datasets. For some program and input distributions, the state-of-the-art neural synthesis models perform quite poorly, often achieving less than 5% generalization accuracy. In our paper, we develop a new methodology for creating training distributions over programs in the DSL to mitigate some of these issues. Moreover, unlike previous works that have ignored considering the distributions over input space, we show that input distributions also play a significant role in determining the synthesizer performance. Our methodology involves defining the distribution over DSL programs and input space using a set of random variables that encodes much of the valuable features that describe the data, e.g. in the Karel domain, the amount of control flow nesting in programs or the number of markers present in the inputs.

Our methodology allows us to identify several specialized distributions over the input space and Karel programs on which the current state of the art synthesis models (Bunel et al., 2018) perform poorly when trained on traditional program and input distributions, and tested on our new distributions. From this, we design new training distributions by ensuring greater uniformity over the random variables in our methodology. By retraining the same architecture on these new training data distributions, we observe a greater ability to generalize, with significant improvements when evaluated on the aforementioned test sets.We also observe similar improvements in the *Calculator* domain as well.

This paper makes the following key contributions:

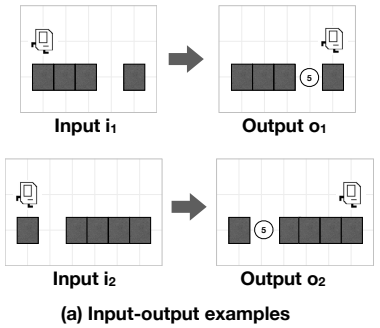
- We propose a new methodology to generate different desirable distributions over the space of datasets for program induction and synthesis tasks.
- We instantiate the methodology for the Karel and Calculator domains and show that model generalization is worse on datasets generated by our technique.
- We then retrain models in both domains and demonstrate that models achieve greater overall generalization performance when trained on datasets generated with our methodology.

2 Related Work

Training models with synthetic data. In certain domains like computer vision and robotics, collecting high-quality real-world training data incurs significant cost, and so many researchers have investigated the use of large amounts of synthetic data. For example, Christiano et al. (2016); Peng et al. (2017); Pinto et al. (2017); Bousmalis et al. (2017) aim to learn robotics policies that compensate for differences between the real world and the simulation. Within computer vision, Shrivastava et al. (2016) demonstrate learning from entirely synthetic images for gaze and pose estimation.

Neural program induction and synthesis. Program induction methods learn differentiable modules such as stack (Joulin & Mikolov, 2015), RAM (Kurach et al., 2016), GPU (Kaiser & Sutskever, 2015), and read-write external memory (Graves et al., 2014) to represent algorithms. Other methods attempt to learn differentiable control flow operations (Gaunt et al., 2016; Neelakantan et al., 2015). These approaches reconstruct outputs given inputs, inferring the underlying algorithms (Guu et al., 2017). Select different algorithms learn directly from program traces rather than from I/O examples (Reed & de Freitas, 2015; Xiao et al., 2018; Cai et al., 2017).

Devlin et al. (2017); Parisotto et al. (2017) use neural program synthesis techniques for learning string editing programs in RobustFill. Similarly, Balog et al. (2016) learn array programs in DeepCoder, and Bhupatiraju et al. (2017) learn to compose API calls. Bunel et al. (2018) apply neural program synthesis to the Karel domain we consider in our paper; we use their architecture and dataset. Many of these approaches report high test performance, but good performance on synthetically-specificed programs need not indicate the model's ability to generalize to arbitrary user-desired programs. In fact, our results show that under certain distributions, these models perform quite poorly. To verify these models appropriately generalize to reasonably complex arbitrary programs, the test set should sufficiently represent the universe of these programs and their specifications. Likewise, to avoid biasing the result, the test set should also draw uniformly from these programs and specifications. For example, for RobustFill, the data generation methodology only sampled programs uniformly from the string DSL, but did not take into account the distribution over input strings such as their length, frequencies of occurrence of regular expressions and their nesting, common words and constants etc.




```
def main():
    while(not(rightIsClear())):
        move()
    turnRight()
    move()
    repeat(5):
        putMarker()
    turnLeft()
    turnLeft()
    move()
    turnRight()
    while(frontIsClear()):
        move()
```

(b) Karel Program

Figure 1: An example Karel synthesis task, where the goal is to synthesize the program in (b) given the two I/O examples shown in (a).

3 Our Data Generation Methodology

Currently, automated data generation focuses in large part on a constructive process, whose parameters can be tuned. We propose a complementary approach in which we perform post-selection on this data to ensure certain properties.

We define a *salient* random variable as one whose distribution in the final dataset is of interest. In the case of program synthesis, there are two kinds of salient variables: $X = \{X_1, \ldots, X_n\}$ and $Z = \{Z_1, \ldots, Z_m\}$, where the random variable X_i denotes a certain important feature of a program in the DSL D and the random variable Z_j denotes a feature for the input space.

In many cases, we can modify our sampling procedure to ensure a desirable distribution of a particular salient variable. However, for some salient variables, it is infeasible to tune the parameters of a given sampling procedure in order to obtain a desired distribution for that salient variable. For example, if we sample programs directly from a context-free grammar, we can end up biasing various salient variables such as program length, degree of nesting, etc. This is a notable problem in both the Karel and Calculator domains.

Furthermore, within the context of program synthesis specifically, there is often an additional challenge: not all inputs are valid for all programs. For example, in the Karel domain, Karel is not allowed to `move` into walls or `pickMarker` in a cell containing no markers. The requirement that the program/input pairs suit each other itself acts as an unpredictable post-selector that makes it difficult to ensure uniformity of salient variables by tuning generation parameters.

Our proposed solution is an algorithm for sampling from a given distribution p while homogenizing (ensuring the uniformity of) a certain salient variable X. Our process is as such: we produce a sample $s \sim p(S)$ and then reject with probability $1 - g(s)$, where

$$g(s) = (P[X = X(s)] + \varepsilon)^{-1}(\min_x P[X = x] + \varepsilon),$$

with the probabilities calculated empirically based on past samples. If we were to set $\varepsilon = 0$, this would ensure that $X(s)$ would be uniform in the resulting distribution (proof in the Appendix, Section 8.2). Unfortunately, we would have no guarantees on runtime. If, however, ε is non-zero, we know that drawing a single sample is possible in $O(\frac{1}{\epsilon})$ calls to the original sampler (proof in the Appendix, Section 8.3, shown to be a reasonable bound in the Appendix, Section 8.4). Increasing ε increases the algorithm's speed, at the cost of allowing $P[X]$ to differ from uniform, reducing performance.

4 Analysis of Existing Models in the Karel Domain

Karel is an educational programming language (Pattis, 1981) where the programmer writes imperative programs with conditionals and loops to produce a sequence of actions for an agent (a robot named Karel) which lives in a rectangular $m \times n$ grid world. For a detailed description of the particular instantiation of the Karel language and input grid specification that we consider, see Figure 4 in the Appendix. The program synthesis task that we consider is as follows: given a set of pairs of input

and output grids $\{(i_1, o_1), \cdots, (i_n, o_n)\}$, find a Karel program π such that executing π on i_1 results in o_1, i_2 results in o_2, and so on. An example Karel synthesis task with the I/O examples and the corresponding Karel program to be synthesized is shown in Figure 1.

In this section, we employ the Karel instantiation of our abstract data generation methodology in Section 4.1 to generate different test datasets. By imposing a more uniform distribution over the salient random variables when generating the I/O specifications and target programs which make up the test set, we observe much lower accuracies of the previous Karel synthesis models (Bunel et al., 2018) compared to the original test set.

4.1 Instantiation for the Karel domain

We devised the following salient random variables to describe the input space in Karel:

- *Grid size*: Dimensions of the grid in which Karel can act.
- *Marker ratio*: Fraction of cells with at least one marker.
- *Wall ratio*: Fraction of cells which contain a wall.
- *Marker count*: Number of markers that are present in a cell containing markers.

For the program space in the Karel DSL, we consider the following random variables:

- *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).

4.2 Changing I/O distributions

We reproduced the encoder-decoder model of Bunel et al. (2018) and trained it using the provided synthetic training set with the teacher-forcing maximum likelihood objective. On the existing test set, our model achieves 73.52% generalization accuracy, slightly higher than the 71.91% accuracy reported in (Bunel et al., 2018). *Generalization accuracy* denotes how often the model's output is correct on the 5 I/O examples shown to the model and the remaining held-out 6$^{\text{th}}$ I/O example.

To test how the model may be sensitive to changes in the I/O examples used to specify the program, we created new test sets by sampling new input grids and running them on each of the programs in the existing test set to obtain new I/O pairs. By keeping the programs themselves the same, we avoid inadvertent changes in the inherent difficulty of the task.

4.2.1 Salient random variables with uniform distribution

We first generated grids such that they would follow a distribution that is as uniform as possible in the salient features in Section 4.1. We used the following procedure to sample each grid: 1) sample the *grid size* (height and width) from $x, y \sim \mathcal{U}\{2, \dots 16\}$; 2) sample the *marker ratio* $r_{\text{marker}} \sim \mathcal{U}(0, 1)$ and *wall ratio* $r_{\text{wall}} \sim \mathcal{U}(0, 1)$; 3) for each cell $(i, j), 0 \leq i < x, 0 \leq j < y$ in the grid, sample $m_{i,j} \sim Bernoulli(r_{\text{marker}})$ and $w_{i,j} \sim Bernoulli(r_{\text{wall}})$; 4) if $m_{i,j} = 1$ and $w_{i,j} = 0$, sample *marker count* $mc_{i,j} \sim \mathcal{U}\{1, \dots 9\}$, otherwise set $mc_{i,j} = 0$; 5) place walls and markers in grid according to $w_{i,j}$ and $mc_{i,j}$; 6) place Karel at a random location (not containing a wall) and with a random orientation. After generating 5 input grids for a given program, we ensure that the program does not crash on any of them and also check whether the 5 input grids exhibit complete *branch coverage*. If either of these conditions are not satisfied, we discard all 5 grids and start over.

On this dataset, the model trained on existing data achieved generalization accuracy of only 27.9%, which was a drop of 44.6pp from the existing test set's generalization accuracy of 73.52%.

4.2.2 Salient random variables with narrow distributions

We further investigated the performance drop 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

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 $Geom(0.5), \mathcal{U}\{1, \dots 9\}$ and $10 - Geom(0.5)$ respectively. See Section 4.2.2 for dataset generation details.

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 generalization accuracy on action-only programs in the existing test set is 99.24%. See Section 5 for details on the Action-Only Augmented model.

	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%

$\mathcal{D}_{\text{marker count}}$ which would be the same for all I/O grids. Then, we follow the procedure below for each grid: 1) sample the *grid size* (height and width) $x, y \sim \mathcal{U}\{10, \dots 16\}$; 2) randomly choose $xy \cdot r_{\text{wall}}$ cells to contain walls, and $xy \cdot r_{\text{marker}}$ cells for markers; 3) sample $mc_{i,j} \sim \mathcal{D}_{\text{marker count}}$ for all cells (i, j) chosen to contain markers; 4) place Karel at a random location (not containing a wall) and with a random orientation. In our experiments, we primarily used 3 different distributions for $\mathcal{D}_{\text{marker count}}$: $Geom(0.5)$ truncated at 9, $\mathcal{U}\{1, \dots 9\}$, and $10 - Geom(0.5)$ which, when sampled, has a value equal to 10 minus a sample from Geom(0.5), truncated at 1.

The results are shown in Table 1 (row 1, "Baseline (%)"). We discovered that the most correlated factor with model performance was the distribution $\mathcal{D}_{\text{marker count}}$. A more negative skew consistently lowered model performance, and this effect was more pronounced at higher values of r_{marker}.

4.3 Changes in the distribution of programs

We will now examine how the existing model can surprisingly fail to perform well at synthesizing certain programs that are different from those in the existing validation and test sets.

Performance on complex DSL constructs. We examined whether or not the model could succeed in synthesizing programs which require nesting of conditional constructs. This was of interest since these programs were relatively rare in the training dataset. We obtained approximations for $r_{\text{wall}}, r_{\text{marker}}$ and $\mathcal{D}_{\text{marker count}}$ for the provided training set, and subsequently 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.

Programs only containing actions. 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 contains actions is intrinsically a much more straightforward task, which a relatively simple search algorithm (such as A*) can perform well.

We performed an experiment using test datasets generated by enumerating action-only programs of various lengths. As there are five actions (`move`, `turnLeft`, `turnRight`, `putMarker`, `pickMarker`), there exist 5^L textually unique action-only programs for length L. We sampled up to 500 unique programs of lengths $1, 2, \dots, 8$. For each program, we generated 10 specifications, each containing 5 I/O pairs. We sampled each I/O pair from the set of all input grids in the existing training data (of which there are 6.7 million), as to match its distribution as closely as possible.

Table 2 shows the results. Remarkably, even though the underlying programs have relatively low complexity, the model's accuracy is lower on every one of these action-only test sets than the existing

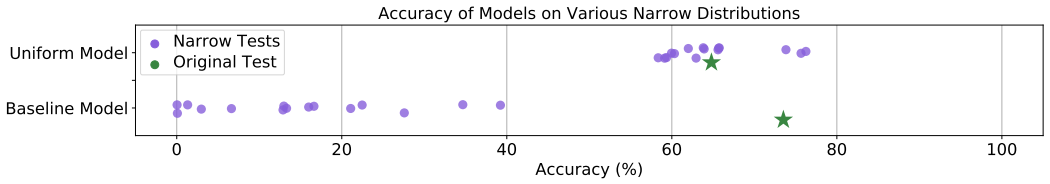


Figure 2: Comparison of generalization accuracies across the datasets given in Table 1 plus performance on the original test set. Both models, denoted *Baseline* and *Uniform* were trained on the same programs. However, the I/O specifications for *Uniform* had homogenized random variables in the way described in Section 4.2.1

provided test set. The generalization accuracy grows as the program length becomes longer, even though those programs should be harder to synthesize.

Among the existing action-only programs in the test set, the model's generalization accuracy on that subset is 99.24%. Given the surprising nature of this result, we investigated the difference between the action-only programs we generated, and those in the existing test set. We found that in the existing training and test sets, all programs contain at least two actions, and also contain at least one `move` action somewhere in the program. These and any undiscovered differences in the distribution of programs seem to have caused the gap in performance.

5 Training the Karel Model on New Datasets

In Section 4, we saw that the existing model performs much more poorly on certain test datasets that we constructed, compared to its performance on the existing test set as reported in Section 4.2. In light of the framework in Section 3, various imbalances of the salient random variables in the existing training data could have caused these gaps in performance. Then a natural solution is to to train using datasets constructed to avoid undesirable skews in the salient random variables, which should hopefully perform better across a variety of distributions.

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 using the procedure described in Section 4.2.1. We trained a model on this data and then evaluated it on the same set of narrow distribution evaluation datasets as mentioned in Section 4.2.2. Figure 2 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 test sets of Section 4.2.2 than the baseline model. Note that the uniform I/O distribution is not simply a union of the tested distributions and is intended to cover all possible input specifications.

Real-world benchmarks. We evaluated both the baseline model and the uniform model on a set of 36 real-world Karel programming problems. This dataset was compiled from the Hour of Code Initiative and Stanford University's introductory computer science course, CS106A, with the problems being hand-designed as educational exercises for students. We found that the baseline model got 4 correct (11.1%) while the uniform model got 7 correct (19.4%). This further demonstrates the uniform model's increased ability to generalize to out-of-distribution datasets, including those which are of interest to humans.

Adding action-only programs. We observed in Section 4.3 that the model fails to do well on either action-only programs or programs with many control-flow statements. In the case of action-only programs, we found that the training data had been pruned to only include programs with at least two actions and at least one `move`, and in the case of programs with complex control flow, we found a similar sparsity in the train set.

As discussed in Section 3, the principled way to counteract this sparsity is to introduce uniformity into a set of salient variables. This methodology allows us to counteract both naturally sparse data

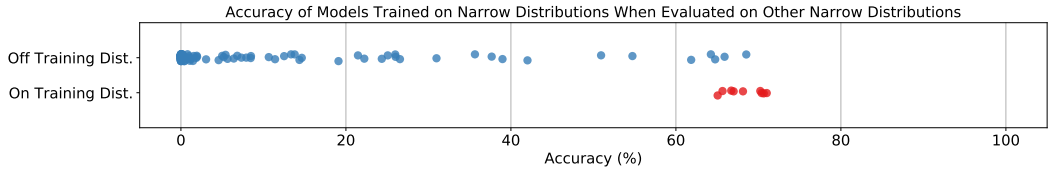


Figure 3: Evaluation of models that were trained on narrow distributions. When evaluated on their own training distribution, they consistently achieved similar results. When evaluated on a different narrow distribution, the performance was rarely similar and usually very low.

(such as complex control-flow) and spurious data pre-processing (such as enforcing programs to have at least two actions).

In our case, we introduce uniformity into the length of action-only programs by synthesizing 20,000 programs of each length 1 to 20, by uniformly selecting tokens from the five action choices and generating I/O with other salient random variables as close to the original training set as possible; we append these new programs to the original dataset to train a new model. Table 2 shows a clear improvement when homogenizing this salient random variable.

Narrowly specified training datasets. As done in for the uniform training dataset, we generated "narrow" training datasets by keeping the same programs as in the existing training data and replacing the I/O pairs with the process from Section 4.2.2.

We trained a variety of models and evaluated them on 12 datasets of different I/O feature distributions. Figure 5 summarizes the results of evaluating each model on each dataset by noting the performance of the model on the narrow dataset of the same type and the outcome on every other narrow dataset. For the models trained on the low variance datasets, we observed that they all consistently achieved between 60 and 70 percent accuracy on their own training distribution; however, the uniform model was able to achieve similar performance (between 57 and 80 percent accuracy). Hence, we conclude that models can be trained to perform well on out-of-distribution data (e.g., the uniform model), while still achieving comparable performance on narrow sub-distributions, as measured against models explicitly trained on such distributions.

6 CALCULATOR

The Calculator task is given as follows: given an expression such as `"5+4*(2+3)"`, compute the result mod 10; in this case, 5. We are interested in the Calculator domain because, while Calculator is not a program synthesis task, creating data for the Calculator problem involves sampling from a context-free grammar. Additionally, Calculator is not as intricate a domain as Karel and thus we can more completely control the environment of data generation with less fear of lurking variables.

Calculator Model. Similar to the work by Zaremba & Sutskever (2014), we implement an LSTM that parses calculator expressions on a character level. We perform a 10-class classification problem using a dense network on the final hidden state of the LSTM. The prediction is correct if it exactly matches the parsed expression value, mod 10.

Distributions of Calculator tasks. We propose 4 distributions for calculator tasks: direct CFG sampling (DCFG), tensor2tensor sampling (T2T), "runs" CFG sampling (RCFG), and balanced sampling (BAL).

Two of our distributions represent reasonable ways in which a researcher might choose to sample data. The first is DCFG sampling, which involves returning a digit with some probability $(1 - p)$, or else recursively sampling two productions and combining them with a $-$, $*$, or $+$, each with probability $\frac{p}{3}$. This corresponds to a direct, weighted, sampling of the CFG for the calculator grammar. The second is T2T sampling[1], which is used by the tensor2tensor library to sample arithmetic expressions in one

[1] https://github.com/tensorflow/tensor2tensor/blob/8bd81e8fe9dafd4eb1dfa519255bcbe3e33c7ffa/tensor2tensor/data_generators/algorithmic_math.py

	Original	Length	Max Depth	Mean Depth	#Operations	#Parens
T2T	83.83%	+4.35pp	+4.24pp	+2.14pp	+1.19pp	+2.32pp
DCFG	78.25%	+3.84pp	+5.92pp	+4.02pp	+6.72pp	+4.51pp

Table 3: Improvements in Calculator performance over unhomogenized distributions when various homogenizations were applied. See Section 6 for details on performance metrics.

of its examples. It involves sampling a depth d, then ensuring that the resulting AST has depth d by forcing a random side of the operation production to be sampled to $d-1$ and the other side to be sampled to a depth $d' \sim \mathcal{U}\{0, 1, \ldots, d-1\}$.

The other two distributions represent potentially difficult or nonstandard problems that might appear in practical environments. RCFG is similar to DCFG sampling but involves increasing the frequency of "runs" of the associative operations picking 2, 3, or 4 productions and then combining them with the given symbol. Balanced sampling involves selecting a depth and then creating an AST that is a balanced binary tree at that depth. Importantly, regardless of sampling technique, redundant parentheses are removed. This is to increase the difficulty somewhat as order of operations needs to be established.

6.1 Salient Variables and Methodology

We use the following salient variables: length (rounded to the nearest even number), number of operations, number of pairs of parentheses, mean parenthesized depth, and maximum parenthesized depth. Parenthesized depth is defined for each digit and refers to the number of nested parentheses it is in. For example in `(1+2)*(3-4)+5`, the 1, 2, 3, and 4 are at depth 1 while the 5 is at depth 0.

We constructed $2 \times (1 + 5)$ distributions in total, corresponding to a total of 2 task distributions, T2T and DCFG, which represent the "natural" sampling techniques a researcher might employ, and $1 + 5$ homogenization strategies, one unhomogenized and five homogenized corresponding to each salient variable with $\varepsilon = 0.025$. We then evaluated each model on a fresh evaluation set sampled from a mixture of the four unhomogenized distributions (T2T, DCFG, RCFG, BAL).

6.2 Results

The original performances and improvements created by homogenizing different random variables can be found in Table 3. On average, homogenizing the DCFG and T2T distributions caused them to increase by 5.00pp and 2.84pp, respectively.

We note that the Calculator domain is much simpler than Karel when considering both input complexity (grid worlds versus arithmetic expressions) and output complexity (a DSL program versus a single digit). Furthermore, the difference in distributions between the naive sampling approaches and the versions with one homogenized random variable are not as different in Calculator as what we observed in Karel (see Table 1 for the dramatic effect of $\mathcal{D}_{\text{marker}}$). We hypothesize that it is this difference in complexity that explains the smaller (but still consistent) effect of homogenizing salient random variables in Calculator as compared to in Karel.

7 Conclusion

We demonstrate that uniform sampling, a widely accepted method for randomly generating input-output examples, has unintended and overlooked distribution flaws in both the Calculator and the Karel domain. These flaws prevent predictive models from generalizing to many simple distributions. To resolve these problems, we propose a robust strategy for controlling and evaluating the bias of synthetic data distributions over programs and specifications by homogenizing salient random variables, such as the number of parentheses in a calculator expression. Equipped with our method, deep networks exhibit an increase in cross-distribution test accuracy, at the expense of a minor decrease in on-distribution test accuracy. We choose program synthesis as a convenient domain, but any domain with synthetically generated data would benefit well from homogenization of salient variables.

REFERENCES

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8, 2013. URL http://ieeexplore.ieee.org/document/6679385/.

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

Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API programmer: Learning to program with apis. *CoRR*, abs/1704.04327, 2017. URL http://arxiv.org/abs/1704.04327.

Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. *CoRR*, abs/1709.07857, 2017. URL http://arxiv.org/abs/1709.07857.

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

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *CoRR*, abs/1704.06611, 2017. URL http://arxiv.org/abs/1704.06611.

Paul F. Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *CoRR*, abs/1610.03518, 2016. URL http://arxiv.org/abs/1610.03518.

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

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL http://arxiv.org/abs/1608.04428.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012. doi: 10.1145/2240236.2240260. URL http://doi.acm.org/10.1145/2240236.2240260.

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11): 90–99, 2015. doi: 10.1145/2736282. URL http://doi.acm.org/10.1145/2736282.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *CoRR*, abs/1704.07926, 2017. URL http://arxiv.org/abs/1704.07926.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, pp. 190–198, 2015.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *ICLR*, 2016.

Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pp. 525–530, 2014.

Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14 (3):151–165, 1971.

Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015. URL http://arxiv.org/abs/1511.04834.

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

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

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. *CoRR*, abs/1710.06537, 2017. URL http://arxiv.org/abs/1710.06537.

Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. *CoRR*, abs/1710.06542, 2017. URL http://arxiv.org/abs/1710.06542.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015. URL http://arxiv.org/abs/1511.06279.

Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Josh Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. *CoRR*, abs/1612.07828, 2016. URL http://arxiv.org/abs/1612.07828.

Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. doi: 10.1007/s10009-012-0249-7. URL https://doi.org/10.1007/s10009-012-0249-7.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018. URL http://arxiv.org/abs/1803.07416.

Richard J. Waldinger and Richard C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pp. 241–252, 1969.

Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. *CoRR*, abs/1802.02696, 2018. URL http://arxiv.org/abs/1802.02696.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL http://arxiv.org/abs/1410.4615.

8 APPENDIX

8.1 THE KAREL DOMAIN

Prog p	:=	`def main():`s
Stmt s	:=	`while`$(b):s$ | `repeat`$(r):s$ | $s_1;s_2$
	|	a | `if`$(b):s$ | `if`$(b):s_1$`else`$:s_2$
Cond b	:=	`markersPresent()` | `leftIsClear()`
	|	`rightIsClear()` | `frontIsClear()`
	|	`not`(b)
Action a	:=	`move()` | `turnLeft()` | `turnRight()`
	|	`pickMarker()` | `putMarker()`
Cste r	:=	$0 \mid 1 \mid \cdots \mid 19$

(a)

`GridWidth`: m
`GridHeight`: n
`Markers`: $\{(i,j,k)_l\}_l$
`Walls`: $\{(i,j)_t\}_t$
`KarelLoc`: (i,j)
`Orientation`: $d \in \{$`N`,`S`,`E`,`W`$\}$
$2 \leq m, n \leq 16; i \leq m;$
$j \leq n; 1 \leq k \leq 9$

(b)

Figure 4: (a) The DSL of Karel programs taken from Bunel et al. (2018), and (b) a declarative specification of the space of valid input worlds for Karel programs.

A declarative specification of the space of valid input worlds to Karel programs is shown in Figure 4(b). As with Bunel et al. (2018), we assume a bound on the input grid size to be $2 \leq m, n \leq 16$. Each cell (i, j) in a grid can either be empty, contain an obstacle (i.e. a wall, specified by the list `Walls`), or contain $k \leq 9$ markers (defined using the list `Markers`). The agent starts at some cell denoted by `KarelLoc` in the grid (which may contain markers but no obstacle) with a particular orientation direction denoted by `Orientation`.

The grammar for the Karel DSL we consider in this work is shown in Figure 4(a). The DSL allows the Karel agent to perform a `move` action to move one step in the grid in the direction of the orientation, actions `turnLeft` and `turnRight` to change its orientation direction, and actions `pickMarker` and `putMarker` to manipulate markers. The language contains `if`, `ifElse`, `while` constructs with conditionals `{front,left,right}IsClear`, `markersPresent`, and their negations. The `repeat` construct allows for a fixed number of repetitions. Note that the language does not contain any variables or auxiliary functions.

8.2 SALIENT VARIABLE HOMOGENIZATION: PROOF OF CORRECTNESS

We wish to show that if $\varepsilon = 0$, the salient variable analysis technique will always result in a distribution where the homogenized salient variable X is uniform. Let the initial sampling distribution be q and the resulting distribution be r. Since we are performing rejection sampling, we have that the probability of selecting any given sample s in our model is

$$P_r[S = s] \propto q(s)g(s)$$

where $g(s) = \frac{\min_x P_q[X=x]}{P_q[X=X(s)]} \propto P_q[X = X(s)]^{-1}$. We thus have that

$$P_r[X = x] = \sum_{s:X(s)=x} P_r[S = s] \propto \sum_{s:X(s)=x} q(s)g(s) \propto \sum_{s:X(s)=x} q(s)P_q[X = X(s)]^{-1} = 1$$

And thus we have that $P_r[X = x] = k$ and is therefore uniform.

8.3 SALIENT VARIABLE HOMOGENIZATION: EFFICIENCY ANALYSIS

In the Salient Variable Homogenization algorithm, we have the probability of not rejecting a given sample as $g(s) = \frac{\min_x P[X=x]+\varepsilon}{P[X=X(s)]+\varepsilon}$. We know that

$$g(s) = \frac{\min_x P[X = x] + \varepsilon}{P[X = X(s)] + \varepsilon} \geq \frac{\varepsilon}{P[X = X(s)] + \varepsilon} \geq \frac{\varepsilon}{1 + \varepsilon}$$

and thus we have that the expected number of tries $t = \frac{1}{g(s)} \leq 1 + \frac{1}{\varepsilon}$. We thus have that in expectation, we need to sample $O(\frac{1}{\epsilon})$ samples from the original distribution to produce one homogenized sample.

8.4 Empirical Effect Of Varying ε

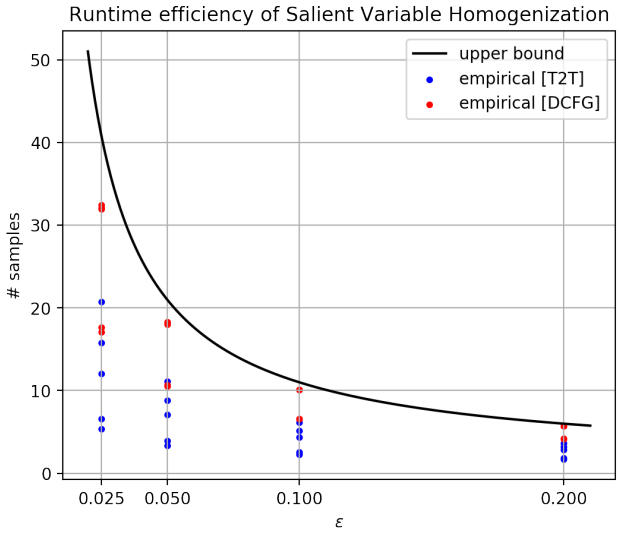


Figure 5: Number of samples required by salient variable homogenization parameterized by an ε before a new sample is returned.

The solid line is an upper bound $\frac{\varepsilon}{1+\varepsilon}$ derived in Section 8.3. Seen in the measured samples for different values of ε, the upper bound appropriately reflects the maximum height of the samples, with very little remaining space on the DCFG dataset and some but not much on the T2T dataset, and is thus a close bound. As the values of $\varepsilon \to \infty$ the bound approaches the limit 1, indicating no samples are rejected by the algorithm.

	Original	$\varepsilon = 0.025$	$\varepsilon = 0.050$	$\varepsilon = 0.100$	$\varepsilon = 0.200$
T2T	83.83%	+2.84pp	+1.31pp	+1.33pp	+2.45pp
DCFG	78.25%	+5.00pp	+4.50pp	+3.54pp	+3.42pp

Table 4: Improvements in Calculator performance with homogenized datasets of various sampling parameters ε. See Section 6 for details on performance metrics.

Increasing the parameter ε has the theoretical affect of causing the homogenized distribution to deviate more from uniform in its salient random variables, as shown in 8.3. In practice, we discover that performance boosts tend to decrease with increasing ε, although the effect was not as pronounced on the T2T dataset, potentially because the unhomogenized T2T distribution is closer to uniform and thus homogenization has a limited effect for any larger ε values.