Data Generation for Neural Programming by Example

Judith Clymo* University of Leeds scjc@leeds.ac.uk
Haik Manukian* University of California at San Diego hmanukia@ucsd.edu
Nathanaël Fijalkow CNRS, LaBRI Alan Turing Institute nathanael.fijalkow@labri.fr
Adrià Gascón Google adriagascon@gmail.com
Brooks Paige UCL Alan Turing Institute b.paige@ucl.ac.uk

Abstract

Programming by example is the problem of synthesizing a program from a small set of input / output pairs. Recent works applying machine learning methods to this task show promise, but are typically reliant on generating synthetic examples for training. A particular challenge lies in generating meaningful sets of inputs and outputs, which well-characterize a given program and accurately demonstrate its behavior. Where examples used for testing are generated by the same method as training data then the performance of a model may be partly reliant on this similarity. In this paper we introduce a novel approach using an SMT solver to synthesize inputs which cover a diverse set of behaviors for a given program. We carry out a case study comparing this method to existing synthetic data generation procedures in the literature, and find that data generated using our approach improves both the discriminatory power of example sets and the ability of trained machine learning models to generalize to unfamiliar data.

1 Introduction

The machine learning community has become increasingly interested in tackling the problem of programming by example (PBE), where the goal is to find a short computer program which is consistent with a set of input and output (I/O) pairs. Methods have been developed that use neural networks either to generate source code directly (Devlin et al., 2017; Parisotto et al., 2016; Bunel et al., 2018) or to aid existing search algorithms (Balog et al., 2017). These papers report impressive empirical performance. However, deep learning methods are data-hungry, and programming by example is not a data-rich regime.

Many neural program synthesis methods are developed targeting domain-specific languages (DSLs), such as the FlashFill environment (Gulwani et al., 2012; Devlin et al., 2017), a custom list processing language (Balog et al., 2017; Feng et al., 2018), or the Karel the Robot environment (Bunel et al., 2018). These languages do not have a large body of human-generated code, nor do they have canonical examples of I/O pairs which correspond to usage in a real-world scenario. As a result, neural program synthesis has turned to generating synthetic data, typically by sampling random programs from some predefined distribution, sampling random inputs, and then evaluating the programs on these inputs to obtain I/O pairs.

The synthetic data is typically used as both the training set for fitting the model, and the testing set for evaluating the model, which is problematic if there is a mismatch between these random examples and potential real-world usage. In the absence of any ground truth this problem of over-fitting is often ignored. In addition, I/O pairs produced by random generation may not be examples that particularly well-characterise a given program. This can affect experimental results through *program aliasing*, where many different possible programs are consistent with the I/O examples (Bunel et al., 2018).

Nearly all existing approaches are based on an implicit assumption that programs and I/O examples used for training should be chosen in a way that is as uniform as possible over the space, despite the fact that many I/O examples are uninformative or redundant, as is well known in the automated software testing community (see e.g. Godefroid et al. (2005)).

* equal contribution

Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS) 2020, Palermo, Italy. PMLR: Volume 108. Copyright 2020 by the author(s).

Contributions. In this paper we consider how to generate sets of I/O examples for training a neural PBE system. In particular we are concerned with the generalizability of neural networks trained for PBE on synthetic data. We present four approaches to data generation in this context, including a novel constraint based method. We show how these different methods can be applied in a case study of the DeepCoder DSL from Balog et al. (2017), an expressive language for manipulating lists of integers. The DeepCoder DSL has several features that make it an interesting and challenging setting for generating synthetic training data: it is capable of displaying complex behavior including branching and looping; the set of valid inputs for a given program can be difficult to define and/or restricted; and the most informative examples are distributed unevenly throughout the space. We train neural networks designed for program generation and assisting a program search using examples produced by the four outlined methods, then use cross-comparison to quantify the robustness of these networks. We also evaluate the degree to which different synthetic data sets uniquely characterise a program, important for addressing the program aliasing problem.

2 Related work

Most approaches to generating I/O examples are based on random sampling schemes, with either a rejection step or initial constraints.

Balog et al. (2017) construct a database of programs in the DeepCoder DSL by enumerating programs up to a fixed length and pruning those with obvious problems such as unused intermediate values. They produce I/O examples for each program by restricting the domain of inputs to a small, program-dependent subset which is guaranteed to yield valid outputs and then sampling uniformly from this set. Feng et al. (2018) targets the same DSL but example pairs are generated by sampling random inputs and seeing whether they evaluate to a valid output. If a sufficient number of valid pairs are not found within a fixed amount of time, the program is discarded. The paper does not state what distribution is used to sample inputs. In both of these papers, this synthetic data is used for both training the model and for evaluating its performance.

Bunel et al. (2018) considers the Karel the Robot domain, and creates a dataset of programs by random sampling from the production rules of the DSL, pruning out programs which represent the identity function or contain trivial redundancies. I/O examples are constructed by sampling and evaluating random input grids. No mention is made of what sampling distribution is used for either the programs or the input grids; the synthetic data is used for both training and testing.

Parisotto et al. (2016) generates synthetic data for the FlashFill domain (Gulwani et al., 2012) by sampling random programs up to a maximum of 13 expressions. The reported performance suggest a very large gap between the synthetic examples (97% accuracy) and the real-world data (38% accuracy). Devlin et al. (2017) is more cautious: performance is evaluated only on the real-world examples, with synthetic data used only for training. The training data is produced by simulating random programs up to a maximum of 10 expressions, and then constructing I/O pairs by sampling random inputs and testing to see whether executing the program raises an exception. Neither paper explicitly specifies any of the sampling distributions.

The papers above all primarily focus on advancing search algorithms or improving heuristics, with little attention to data generation; the only exception we are aware of is Shin et al. (2019), which specifically proposes a method for improving synthetic example generation. Their approach requires first defining a set of "salient variables" for the domain: these take the form of a mapping from a synthetic training example to a discrete value. They then define a sampling strategy for examples which aims to maximize their entropy, by generating a training dataset which is overall approximately uniform over all the combinations of the different discrete salient variables. Particular attention is paid to the Karel the Robot domain, and the generation of synthetic data for the algorithms in Bunel et al. (2018).

When there are many salient variables the discrete domain can be large. However, the number of I/O examples needed for each program is typically quite small. In the Karel the Robot domain the input space is not constrained by the programs so it is natural to enforce uniformity across salient variables over the whole set of inputs. Where the program can significantly affect the valid input space the approach is not appropriate. The contributions we make to this task are complementary to those of Shin et al. (2019), which in this domain could still be beneficially applied in selecting a set of synthetic programs, but would be difficult to adapt to generating meaningful inputs.

3 ML-aided Programming by Example

Figure 1 outlines the general approach to programming by example we take in this paper, providing a framework which can be used to understand methods and settings employed in recent work. The problem is de-

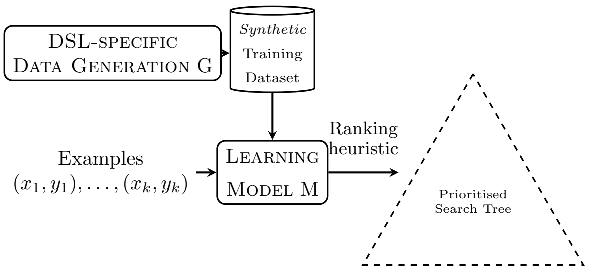

DSL-specific Data Generation G
Synthetic Training Dataset
Examples $(x_1, y_1), \ldots, (x_k, y_k)$
Learning Model M
Ranking heuristic
Prioritised Search Tree

Figure 1: ML-aided programming by example

fined in terms of a concrete DSL that takes as input a set of examples $(x_1, y_1), \ldots, (x_k, y_k)$, and synthesizes a program mapping inputs x_i to the corresponding outputs y_i. At the core of the approach is a search algorithm that explores the space of all programs up to a certain predefined length according to a ranking function. This heuristic ranking function is derived from the output of an ML model M that inspects the input $(x_1, y_1), \ldots, (x_k, y_k)$. Crucial in this approach is the training dataset of M. This dataset is obtained using a DSL-specific generation procedure G that generates (a) programs and (b) a small set of inputs for those programs.

Hence, to instantiate a concrete approach to ML-aided programming by example one needs to define (i) the DSL, (ii) the data generation procedure G, and (iii) the ML model M and corresponding search procedure. As mentioned above, previous work focuses primarily on (iii). Here we instead focus on (ii), the training data generation step, and investigate its effect in concrete instances of the complete pipeline.

3.1 Domain specific language

The DeepCoder DSL (Balog et al., 2017), which we will use as our running example, consists of high-level functions manipulating list of integers; a program is a sequence of functions which take as input any of the inputs or previously computed values. The DSL contains the first-order functions Head, Last, Take, Drop, Access, Minimum, Maximum, Reverse, Sort, Sum, and the higher-order functions Map, Filter, Count, ZipWith, Scanl1. The higher-order function Map can be combined with `(+1)`, `(-1)`, `(*2)`, `(/2)`, `(*(-1))`, `(**2)`, `(*3)`, `(/3)`, `(*4)`, `(/4)`. The two higher-order functions Filter and Count use predicates `(>0)`, `(<0)`, `(%2==0)`, `(%2==1)`. Finally, ZipWith and Scanl1 are paired with `(+)`, `(-)`, `(*)`, Min, Max. The lists in a program are of a predefined length (20 in the experiments of Balog et al. (2017)) with values in $[-255, 256] \cup \{\perp\}$, with $\perp$ denoting an *undefined* value. The semantics of the DSL assume some form of bound checking, as indexing a list out of its bounds returns $\perp$, and over(under)-flow checking, as values outside of the range $[-255, 256]$ evaluate to $\perp$. As we will see, this impacts data generation by constraining the domain of valid inputs. This DSL is remarkably expressive: it can express non-linear functions via repeated integer multiplication, control flow (if-then-else statements can be encoded by means of Filter), and loops over lists (by means of the higher-order functions). Moreover, the higher-order functions like Reverse, Sort, ZipWith and Scanl1 allow to encode surprisingly complex procedures in just a few lines, such as the four-line example shown in Figure 2.

4 Data Generation Methods

We propose three new approaches for the data generation step. The first is based on a probabilistic sampling of the input space, while the other two treat data generation as a constraint satisfaction problem. We develop a general framework for this approach and show how additional constraints can be used to impose variation in the I/O pairs produced.

4.1 Sampling-based approaches

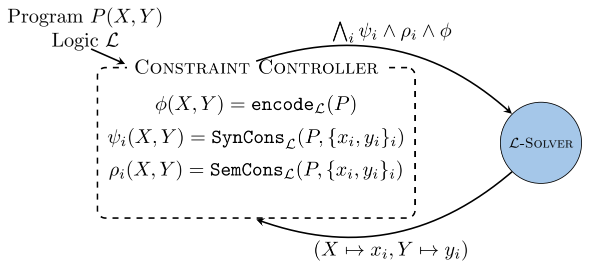
We use the approach from Balog et al. (2017) as a baseline. Programs are evaluated in reverse to derive a 'safe' range for input values that is guaranteed to produce outputs in a target range. Values are then sampled uniformly from the safe input range to create the input(s), and the output is calculated by evaluating the program. If the safe range for some input is empty or a singleton, then the program is discarded. In reporting results of our experiments, we refer to this method of data generation as "Restricted Domain".

The purpose of the reverse propagation of bounds is to exclude all non-valid inputs from the sampling. However, some valid inputs are also excluded. Instead, inputs can be sampled from an over-approximation of the set of valid inputs or by using a probability distribution that prioritises parts of the input space where valid inputs are known to be common. Inputs are then rejected if they are unsuitable for the program being considered.

We observe that in the DeepCoder DSL, when the output range is bounded, small input values are compatible with more programs than large values. Sampling with a bias towards small values means that suitable inputs are found with high probability for all programs within a fixed number of attempts n. If suitable inputs for a program remain common outside of the safe range identified by back propagation of bounds then the gain from allowing these to be included could be significant.

```
a ← [int]                 An input-output example:
b ← SORT a                Input:
c ← FILTER (>0) b         [-17, -3, 3, 11, 0, -5, -9, 13, 6, 6, -8, 11]
d ← HEAD c                Output:
e ← DROP d b              [-5, -3, 0, 3, 6, 6, 11, 11, 13]
```

Figure 2: An example program in the DSL that takes a single integer array as its input.



Program $P(X,Y)$
Logic $\mathcal{L}$
$\bigwedge_i \psi_i \wedge \rho_i \wedge \phi$
CONSTRAINT CONTROLLER
$\phi(X,Y) = \texttt{encode}_{\mathcal{L}}(P)$
$\psi_i(X,Y) = \texttt{SynCons}_{\mathcal{L}}(P, \{x_i, y_i\}_i)$
$\rho_i(X,Y) = \texttt{SemCons}_{\mathcal{L}}(P, \{x_i, y_i\}_i)$
$\mathcal{L}$-SOLVER
$(X \mapsto x_i, Y \mapsto y_i)$

Figure 3: Our constraint-solving based approaches are instances of the above model. A constraint controller adaptively produces problems for the solver, whose solutions correspond to new input-output pairs to be added to the training dataset

To produce a sample, we first fix a random length of the input (uniformly at random) and generate a value r from an exponential distribution i.e. with probability $P(r) \sim \text{Exp}(r, \lambda) = \lambda e^{-\lambda r}$. The input values are selected uniformly at random (with replacement) from the range $[-r, r]$, then evaluated on the program. If the output of the program is within the desired range the pair is accepted, otherwise a new value of r is sampled and the process repeated.

Varying the choice of λ modifies the strength of the bias, affecting both the number of attempts that must be allowed and also the similarity of examples generated. In our case, we chose $\lambda = 0.001$ and $n = 500$. This method will be referred to as “Non-Uniform Sampling” in the experiments section.

4.2 Constraint-solving based approaches

Sampling methods are not well suited to navigating valid input spaces that are sparse or irregular, and in a language capable of displaying complex behavior the most informative examples may also be rare. A constraint solver is able to find suitable inputs in parts of the input space that are almost always excluded in the sampling methods. In this section we present a methodology to synthesize a set of examples that relies on constraint solving.

The general procedure is presented in Figure 3 and consists of a feedback loop between a constraint controller and a solver. The former adaptively produces problems for the solver, whose solutions correspond to new I/O pairs to be added to the training dataset. The approach is parametrised by a choice of a logic $\mathcal{L}$, and assumes a solver, denoted $\mathcal{L}$-Solver, that can decide $\mathcal{L}$, i.e. find values for the variables in any formula ϕ from $\mathcal{L}$ for ϕ to evaluate to true, or report “unsatisfiable” if such values don’t exist. We require encoding and decoding procedures $\texttt{encode}_{\mathcal{L}}, \texttt{decode}_{\mathcal{L}}$ such that $\texttt{encode}_{\mathcal{L}}$ takes a program P with input variables X and output variables Y, and produces a formula whose satisfying assignments can be translated by `decode` into concrete input-output pairs of the program P. For simplicity, we assume that the domain of the variables in $\phi_{\mathcal{L}}$ and P is the same in Figure 3 and omit the decoding step.

The simplest data synthesis procedure consists of the controller simply calling the solver iteratively to collect a sequence of input output pairs $(x_1, y_1), \ldots, (x_n, y_n)$ for P. Note that for DSLs like the one presented above, a complete decision procedure might be too much to ask, but for our purposes soundness is sufficient. More concretely, $\texttt{encode}_{\mathcal{L}}$ should be constructed so that, for each program P in the DSL the following holds:

$$\forall XY : \texttt{encode}_{\mathcal{L}}(P) \Rightarrow P(X) = Y$$

This procedure, as described so far, is not very helpful, as nothing prevents the solver from returning the same pair (x, y) at every iteration. This can be easily avoided by the controller imposing the additional constraint $X \neq x_i \vee Y \neq y_i$ after every iteration. This is the most basic form of additional constraint that we consider. More generally, our approach considers two additional sets of constraints, which we call *syntactic constraints*, denoted $\texttt{SynCons}_{\mathcal{L}}$ and *semantic constraints*, denoted $\texttt{SemCons}_{\mathcal{L}}$. While both of this constraints are conditions on I/O pairs – either for each line of the program or the whole program – the former consist of simple equality and inequality checks such as “input and output should be different”. In contrast, semantic constraints are more powerful – and costly to enforce – as they enforce predicates on input output pairs such as “the maximum value of the input should be smaller than the maximum value of the output”.

Table 1: Constraints and predicates used in Syntactic and Semantic constraints used in our constraint-based input generation algorithms. Constraints and predicates are depicted as conditions on the I/O pairs $(x_1, y_1), \ldots, (x_5, y_5)$ generated by the constraint-based approaches. For programs taking more than one input all constraints are applied to each of the inputs. x_i^l is the intermediate program output of line l given input x_i.

$\texttt{SynCons}_{\mathcal{L}}$ for "Constraint Based" and "Semantic Variation"	
$c1 := \forall i : x_i \neq y_i$	*Output does not match input.*
$c2 := \forall i : x_i \neq [\,]$	*All inputs are not empty.*
$c3 := \forall i \neq j : x_i \neq x_j$	*No duplicate inputs in a set.*
$c4 := \forall i \neq j : y_i \neq y_j$	*No duplicate outputs in a set.*
$c5 := \forall i : \lvert x_i \rvert > B$	*Input length larger than random bound B.*

Predicates in $\texttt{SemCons}_{\mathcal{L}}$ for "Semantic Variation"	
$p_1 := \max(y_i) > \max(y_{i-1})$	*Increase maximum.*
$p_2 := \min(y_i) < \min(y_{i-1})$	*Decrease minimum.*
$p_3 := \lvert x_i \rvert > C$	*Input length larger than C.*
$p_{l,4} := head(x_i^l) \neq head(y_i^{l-1})$	*Change in head.*
$p_{l,5} := last(x_i^l) \neq last(y_i^{l-1})$	*Change in last.*
$p_{l,6} := \lvert x_i^l \rvert \neq \lvert y_i^{l-1} \rvert$	*Change in length.*
$p_{l,7} := max(x_i^l) \neq max(y_i^{l-1})$	*Change in maximum.*
$p_{l,8} := min(x_i^l) \neq min(y_i^{l-1})$	*Change in minimum.*

The next two concrete approaches to data generation are instances of the above scheme, and thus correspond to concrete choices for $\mathcal{L}$, `encode`, `decode`, $\texttt{SynCons}_{\mathcal{L}}$, and $\texttt{SemCons}_{\mathcal{L}}$, and the corresponding variations in the behavior of the constraint controller. In our implementation we use Z3 (de Moura and Bjørner, 2008) as a back-end solver. Z3 implements incremental solving, which allows to push and pop constraints into an existing formula while preserving intermediate states, a crucial aspect of the implementation of a controller. As per $\mathcal{L}$, we experimented with the theory of non-linear arithmetic, for which Z3 implements incomplete procedures, and the theory of bitvectors.

4.2.1 Simple constraint solving

Our third data generation method uses a constraint solver to produce valid examples with only minimal extra guidance. Hence, in this case $\texttt{SemCons}_{\mathcal{L}}(P, S) = \emptyset$. $\texttt{SynCons}_{\mathcal{L}}$ are given in Table 1 by means of a list of constraints $c_1, \ldots, c_5$. At the ith iteration the pair (x_i, y_i) is obtained from the solver as a satisfying assignment of the constraints $\bigwedge_i \psi_i \wedge \phi$ (as shown in Figure 3), where $\psi_i = \bigwedge_{i=1}^{5} c_i$. This guarantees that synthesized examples will satisfy these conditions if possible.

4.2.2 Constraint solving to generate varied examples

Our fourth method is inspired by program verification approaches such as predicate abstraction and CounterExample-Guided Abstraction Refinement loop (CEGAR) (Clarke et al., 2003). This method uses constraints more aggressively, and in particular semantic constraints, by implementing an adaptive constraint controller. The syntactic constraints used in this case are as in the previous method, so we focus on describing the semantic constraints, and the adaptive behaviour of the controller.

The controller first uses sampling to generate a small set of examples, and then keeps track of the evaluation of predicates $p_1, \ldots, p_3$ for the output of the program and $p_4, \ldots, p_8$, *for every line of the program* (see Table 1) for each of the inputs found so far. For example, for 5-line programs this corresponds to storing a 5×5 Boolean matrix M^k for each input x_k so far. The controller also maintains a record of constraint combinations that result in unsatisfiable problems so these can be avoided in subsequent calls. The high-level idea is as follows: firstly, by monitoring any bunching of previous examples the controller can recognise programs with valid examples tending to bunch in one part of the I/O space (in this DSL, bunching is always towards zero, so monitoring maximum and minimum is a good proxy) and force the constraint solver to seek out examples in parts of the space where valid examples are relatively rare; secondly by monitoring which of the predefined behaviours $p_4, \ldots, p_8$ are satisfied for each program line, by each of the inputs x_i collected so far, the controller can detect sets inputs that do not sufficiently exercise internal lines of the program, and hence send a weak signal regarding the presence of the function at that line to the output. Besides recording which behaviors are exhibited by each line, the controller can impose a given behavior by asserting a specific predicate (or its negation) as part of the semantic constraints for that iteration. More specifically, the controller defines the semantic constraint of iteration i as $\texttt{SemCons}_{\mathcal{L}}(X, Y) := (\forall k < i : \exists n, m : (p_{n,m}(X, Y) \neq M^k_{n,m}))$, where $p_{n,m}$ is a Boolean formula encoding that the nth line of the program satisfies the mth predicate on input X. This constraint enforces that in any valid assignment $X \mapsto x_i, Y \mapsto y_i$ obtained by the solver, input x_i will induce a behavior that differs with each combination of behaviors induced by inputs found so far in at least one line of the program. This makes the intuitive goal of finding "a varied set of inputs" precise.

Choosing predicates. The choice of features to monitor is specific to the DSL and based on abstrac-

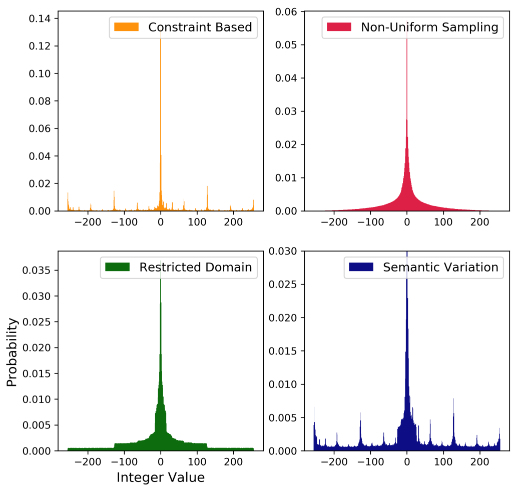


Figure 4: Histograms of the input values found in training sets generated with the four data generation methods considered. Note the large variation in y-axes.

tions of the actions of its constituent functions. In this DSL we are working with lists of integers defined by their length, the set of values contained, and the order in which those values appear. The functions of the DSL can change all of these features. Our aim is that if the program is capable of changing the values that appear in the output compared to the input then there should be an I/O example that demonstrates this, if it is capable of reordering the elements this should also be shown, and so on. The maximum, minimum, first, and last elements of a list act as simplified indicators for changes in the values and order of the list and are the same features used in Feng et al. (2018) to assist the search procedure for this DSL.

We have referred to this data generation method as “Semantic Variation” in the experiments.

5 Experiments

We consider two choices of ranking heuristics; one exactly following Balog et al. (2017), and the other a natural extension based on recurrent neural networks, following a sequential generation paradigm (Devlin et al., 2017; Bunel et al., 2018).

The four synthetic data generation methods discussed above are compared in each of these settings. In particular, we investigate how neural networks trained with data from one method perform at test time on data generated by another. Code for replicating these experiments is available online [0]. For a fair comparison of the data generation methods, we use the same split of programs into training and testing examples. Figure 4 shows how input values are distributed in data generated by each method.

[0]`https://github.com/jclymo/DataGen_NPBE`

The DeepCoder heuristic. The ranking function used in Balog et al. (2017) estimates, for each of the 38 functions in the DSL, the conditional probability that the function ever appears in the program. The predicted probabilities provide a ranking for a depth-first search.

The recurrent neural network heuristic. In addition, we consider an extension to DeepCoder which is loosely inspired by the recurrent neural network architectures used for program generation in other DSLs (Devlin et al., 2017; Bunel et al., 2018). Instead of training the network to output a single set of probabilities, we train a network to output a sequence of probabilities, conditioned on the current line of the program.

We do this by modeling the sequence of lines with a long short-term memory (LSTM) network (Hochreiter and Schmidhuber, 1997). To ease comparison, we leave the majority of the network architecture unchanged; the only modification is the penultimate layer is replaced by an LSTM. The result is a network which takes as arguments not just the I/O examples, but also a target “number of lines”, and then returns estimates of probabilities that a function occurs on a per-line basis, rather than program-wide. Complete architectural details for both networks are in the appendix.

5.1 Cross-generalization of different methods

The plots in Figure 5 show the proportion of programs for which every line is included within the top k predictions given by a neural network. If functions actually present in the program are ranked highly by the network, this will accelerate the runtime of any corresponding guided search. The baseline is given by a fixed ordering reflecting the relative frequency of each function in the set of programs used for training. All networks performed better than the baseline on all test sets, showing that the network is able to generalise beyond its training setting. Test data is always most accurately interpreted by the network trained on data generated through the same process; in fact, the four networks performed almost identically to each other when tested on their own data. However, the ability of each network to transfer to foreign data, from a different generation process, varied significantly. The two sampling methods (Restricted Domain, Non-Uniform Sampling) in particular found the data generated by an SMT solver difficult, while the

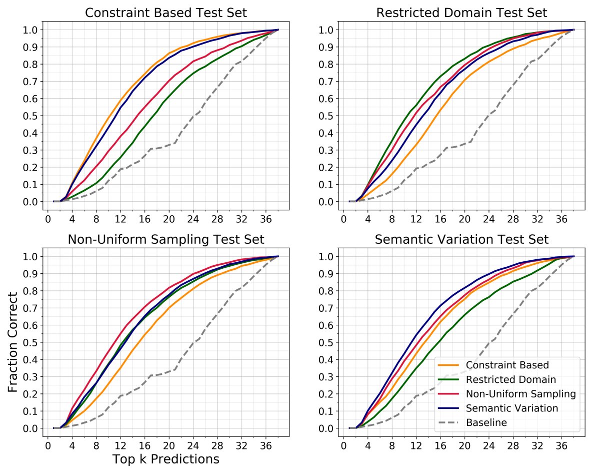


Figure 5: Each network predicts the functions in a given test set program. We report the fraction of programs for which all constituent functions are contained in the top-k predictions. The networks are trained on one mode of generation and tested on all the others. Each line represents the performance of a training set, defined in the legend. From the performance gaps, we see that the network trained on Semantic Variation data appears more robust to a change in test data generation method.

Table 2: Total area under top-k curves, across all approaches to test data generation.

Method	Total AUC
Semantic Variation	126.11
Non-uniform Sampling	123.14
Constraint Based	118.24
Restricted Domain	114.21

loss from the constraint-based methods to the sampling data was smaller, though still marked.

The sum of the area under the curves indicates the robustness of the networks, with the Semantic Variation method having the largest area; values shown in Table 2.

5.2 Program Aliasing

We measure how well the generated example sets characterise their target program by taking a sample of programs and I/O sets generated by each method, and searching for programs no longer than the intended program which matched the given examples. If an alternative program of the same length was recorded across all test sets then this could be due to logical equivalence, and these examples were excluded.

Table 3: Errors due to program aliasing. *Error* corresponds to programs which are correct on the test data, but differ from the original generating program; the predicted programs tend to be *shorter*, and often strictly *contained* within the target program.

Method	% Error	% Shorter	% Contained
Semantic Variation	4	2	2
Constraint Based	6	4	3
Non-uniform Sampling	6	5	4
Restricted Domain	15	12	9

The Restricted Domain method had the highest rate of ambiguous example sets with 15% of the sets tested able to be satisfied by an alternative (not equivalent) program, compared to a 4% error rate on examples generated by the Semantic Variation method. Some errors were very subtle, confusing programs that were logically different only on inputs with a specific form. Others were due to difficulty distinguishing the functions that output integers: in a set of only five examples it is relatively common that the maximum of every list is also at the same position, for example.

The results are summarised in Table 3 which shows the percentage of example sets where an alternative program was found and also indicates whether this alternative program was strictly shorter and strictly contained within the target program (i.e. the target program included some function had no discernible effect for any of the provided inputs).

5.3 Performance on Independent Test Sets

We sought to test the four networks on data that is not generated by a machine and created a small handcrafted test set of examples on programs of length two to five. The performance of the neural networks is shown in Figure 6, in the right-most plot. The test set is clearly too small to draw any serious conclusions, but the good performance of constraint based approaches is encouraging.

As an alternative to this, we fixed five inputs and ran a set of programs on these same five inputs, keeping those for which the resulting example set could not be satisfied by another program of the same or smaller length. Because some functions require mainly small values in the inputs we made one test set by this method which had almost all input values between -10 and 10, and a second set with many more large values. This approach ensures the inputs alone are relatively uninformative, forcing the networks to derive their predictions from the relationship between the inputs and outputs. The results of this experiment are also shown in Figure 6 (left and centre), again giving the propor-

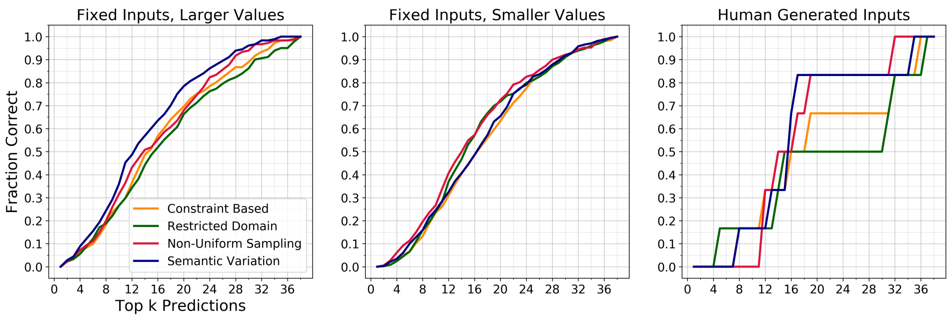


Figure 6: The fraction of programs contained in the top-k predictions of trained networks on hand-crafted examples. Left and middle figures show performance on problems sets with fixed inputs with large and small average values respectively. Right figure shows performance on a small set of human generated examples.

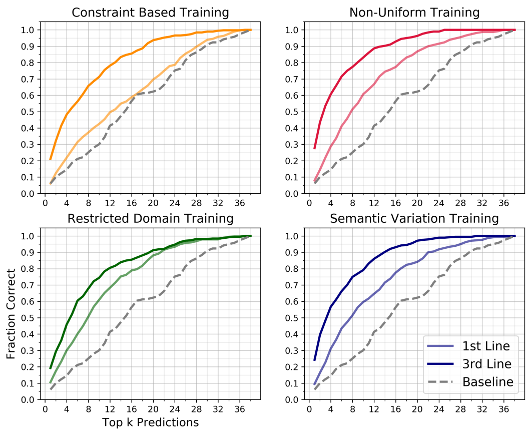


Figure 7: First and last line predictive accuracy of the recurrent architecture on our test set with fixed inputs.

tion of example sets for which the whole program was contained in the top k predictions.

The networks all performed better than random, showing their predictions are not purely reliant on learning something about how the examples are generated. We also observe a difference in performance depending on whether the examples contain mostly small or mostly large values.

Separately, we evaluate the performance of the recurrent neural network architecture, detailed in the appendix, over this difficult set of examples. The same RNN model was trained over sets generated with the four different methods, as for the feed-forward networks. We plot the fraction of times the RNN was correct about the line appearing in the top-k predictions for that line in Figure 7. We see that the last line is easier to predict than the first, for all data generation methods, matching the intuition that as more functions are applied to the data, more information is lost about lines that occurred earlier in the program.

Overall, when compared to baseline, some sets struggle more than others, but much like the feed-forward case, there is no strategy that has a commanding edge over the others.

However, in longer programs than considered in this paper, we imagine that the RNN architecture could more substantially assist a search than the feed-forward networks. The network considered here provides an ordering for all lines at the start of the search. A more advanced approach could update its prediction for the next line given the evolving state of a partial program; we leave this to future work as it requires careful consideration of runtime costs: Repeated re-evaluation of an RNN inside the inner loop of the search algorithm could be inefficient relative to simply running more iterations with a cheaper heuristic, whereas the per-line ranking used here has identical search-time cost as the DeepCoder heuristic.

6 Discussion

All the methods of generating data that we have considered were useful in training the neural network, and each of the four trained networks displayed the ability to generalise to unfamiliar testing data. Conversely, all networks displayed a preference for testing sets generated in the same way as the data used for training, demonstrating over-fitting to the data generation method to some degree.

Constraint solving proved an effective way of discovering examples that were out of reach to sampling methods, and simple local features were useful in creating example sets that characterise the target program well and reduce the occurrence of ambiguous examples. We also saw that the more informative training data in the Semantic Variation method increased the resilience of the neural network against unfamiliar test scenarios.

Acknowledgments

Work started when JC and HM were at the The Alan Turing Institute for the Summer 2018 Internship Programme, NF and AG were at the The Alan Turing Institute and Warwick University, and BP at the The Alan Turing Institute and University of Cambridge. This work is was supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1, by the UK Government's Defence & Security Programme in support of the Alan Turing Institute. We would additionally like to thank Matej Balog for his helpful assistance and feedback.

References

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2017). Deepcoder: Learning to write programs. In *International Conference on Learning Representations*.

Bunel, R., Hausknecht, M. J., Devlin, J., Singh, R., and Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. *CoRR*, abs/1805.04276.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794.

de Moura, L. M. and Bjørner, N. (2008). Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., and Kohli, P. (2017). Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pages 990–998.

Feng, Y., Martins, R., Bastani, O., and Dillig, I. (2018). Program synthesis using conflict-driven learning. In *Conference on Programming Language Design and Implementation*, pages 420–435.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM.

Gulwani, S., Harris, W. R., and Singh, R. (2012). Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. (2016). Neuro-symbolic program synthesis. In *International Conference on Learning Representations*.

Shin, R., Kant, N., Gupta, K., Bender, C., Trabucco, B., Singh, R., and Song, D. (2019). Synthetic datasets for neural program synthesis. In *International Conference on Learning Representations*.