General Program Synthesis using Guided Corpus Generation and Automatic Refactoring

Alexander Wild and Barry Porter
a.wild3@lancaster.ac.uk b.f.porter@lancaster.ac.uk

Lancaster University

Abstract. Program synthesis aims to produce source code based on a user specification, raising the abstraction level of building systems and opening the potential for non-programmers to synthesise their own bespoke services. Both genetic programming (GP) and neural code synthesis have proposed a wide range of approaches to solving this problem, but both have limitations in generality and scope. We propose a hybrid search-based approach which combines (i) a genetic algorithm to autonomously generate a training corpus of programs centred around a set of highly abstracted hints describing interesting features; and (ii) a neural network which trains on this data and automatically refactors it towards a form which makes a more ideal use of the neural network's representational capacity. When given an unseen program represented as a small set of input and output examples, our neural network is used to generate a rank-ordered search space of what it sees as the most promising programs; we then iterate through this list up to a given maximum search depth. Our results show that this approach is able to find up to 60% of a human-useful target set of programs that it has never seen before, including applying a clip function to the values in an array to restrict them to a given maximum, and offsetting all values in an array.

1 Introduction

The ever-increasing complexity of writing software – in design, implementation, and ongoing maintenance – has led researchers to consider how programs can be synthesised automatically from a given specification. This could allow system designers to operate at a higher level of abstraction, defining and verifying functionality rather than implementing the fine details, and also has the potential to allow non-programmers to create custom software.

The state of the art in code synthesis has generally considered the problem for domain-specific languages, such as string manipulation, and also tends to restrict the scope of the problem to programs without loops. DeepCoder, for example, uses these restrictions to demonstrate that neural network training over a randomly sampled corpus can find speed versus exhaustive search for a simple language [1]. FlashFill, meanwhile, demonstrates that inductive programming

can follow user examples to propose possible functions for Excel data transformations within a limited set of operators [8]. Despite these promising results, the limitations of domain specificity and linear logic result in significant restrictions on the kinds of program that can be constructed by code synthesis.

We propose a programming-by-example approach which uses neural-network-based program prediction to operate on a simplified *general-purpose* programming language, with a current focus on integer manipulation, which is capable of producing functions that contain loops and conditional branches. The user is required to supply up to 10 input/output examples which describe the program they wish to create, and programs generated in our intermediate language can be directly and automatically converted to Java or C code. Our neural network is trained on a corpus of synthetic, self-generated examples, the initial population of which is biased using one sample human-useful program. When given a new I/O target pair, the neural network is used to generate a search space which we exhaustively iterate through to a given search depth to find a matching program. In detail, our approach works as follows:

- We use a genetic programming approach to generate a training corpus of programs, based on a seed program which reflects some of the common abstract features believed to be useful in human-required programs. This seed program could be supplied by a human as an over-specified initial program.
- We train a neural network with the resulting corpus, such that the input layer is provided with input/output examples, and the output layers must generate the corresponding program by selecting one line of code per output layer. The neural network is able to both recognise programs that it has already seen and infer programs that it has not.
- We use a technique that we term *automated corpus refactoring* in which the neural network re-trains itself by adjusting its own training corpus based on the kinds of programs it was able to locate from that corpus; we demonstrate that this technique can provide significant improvement in the capabilities of the system to find more unseen programs.

Our results show that we are able to automatically generate 60% of a target corpus of unseen programs based only on 10 I/O examples, including counting how many of a specific value appear in an array, and shifting the contents of an array left or right. We believe that our work is the first to demonstrate that a neural network can be trained to output general-purpose programs that include loops and branch statements, starting only from an automatically generated-corpus based a small set of abstract features that useful programs tend to have. We provide all of the source code for our system, along with instructions on how to repeat our experiments[1].

In the remainder of this paper we first survey related work in Sec. 2, then present our approach in Sec. 3. In Sec. 4 we evaluate our system on both abstract program learning and a specific set of human-useful programs such as searching and array reversal. We then conclude and discuss future work in Sec. 5.

[1] https://bitbucket.org/AlexanderWildLancaster/automaticrefactoringsynthesis.git

2 Background

Program synthesis has long been studied in computer science; in this section we discuss the most relevant research in genetic programming, inductive programming, and neural code synthesis and imitation.

Genetic programming (GP) applies a paradigm of mutation and crossover, seen in biological reproduction, to source code in order to formulate a particular program. A wide range of research has examined topics from improving efficiency to the ability to navigate noisy landscapes and generality of solution [3]. The genetic tools provided by this research have also shown adoption in real-world commercial applications in the sub-field of genetic improvement [12] (GI). However, despite its successes, there are also clear limitations in its use for synthesising programs starting from no initial code. The work "Why We Do Not Evolve Software? Analysis of Evolutionary Algorithms" [16] presents arguments against the current state of the art in genetic algorithms, and the work "Neutrality and Epistasis in Program Space" [13] explores why this may be. Specifically, GP and hence GI rely on finding paths in the fitness landscape of program space from a starting position to the desired functionality. If a program's functionality precludes this incremental path-finding, perhaps because it is a function which cannot 'partially succeed' and must be fully implemented to show success, genetic methods cannot navigate towards it in program space and instead must rely on pure chance to find it. This is more likely to occur in the cases this paper investigates, which have very low numbers of user provided specification-examples, and therefore very low granularity in terms of success/failure metrics. This work focuses instead on using neural techniques to interpolate between learned and recognised functionalities within program space, which does not require a navigable fitness landscape. We show that genetic methods remain critical, however, in the generation of the training corpus used by the neural network, to guide the exploration of program space in a humanly-useful direction.

Inductive programming has been successfully used for a variety of code synthesis tasks, most notably in the FlashFill approach to spreadsheet function generation [8]. In this work, a set of examples is provided by a user, and a sequence of inductive logic passes are applied to incrementally reduce the search space of possible programs which match the examples in a broadly similar way to SMT solving [4][5]. This approach depends heavily on the use of a highly restricted and specialised language over which to search, often with the inductive logic passes being designed specifically with that language in mind. These approaches can synthesise functions very quickly and without training data, but rely on carefully crafted programming languages with associated inductive logic rules, making them hard to generalise to a broader class of synthesis problems.

Neural networks have been applied to the code sythesis problem in two different ways: imitation and synthesis. Neural program imitation works by encoding a program itself as a set of weights in a neural network – literally training a neural network to imitate a program. This has been demonstrated in work such as the Neural Turing Machine [6], the Neural GPU [10] or the Differentiable Neural Computer [7]. These examples show that, from a large number of pure

I/O examples and with no prior knowledge of what any other program looks like, a resulting 'program' can be learned which has high accuracy though remains probabilistic. The main drawbacks are the volume of I/O examples needed (tens of thousands) which arguably are no easier to generate than the algorithm itself; the lack of generality such that the encoded program can correctly operate on longer input lengths that those it was originally trained on; and the lack of scrutability since the program cannot be output as conventional source code, instead being encoded opaquely within the weights of a neural network.

Neural program synthesis, by comparison, trains a neural network on a set of programs by showing it the source code and corresponding I/O pairs, then attempts to generate the source code for unseen programs by issuing new I/O pairs. This is usually done by having the neural network identify a search region in which the program is likely to appear (for example by selecting which operators are most likely) and then searching exhaustively through this region. This approach has the benefit that the neural network outputs source code which can be examined, and that generated programs are both deterministic and tend to generalise across different input sizes. The downside is that a training corpus must be generated which is in some way informative of reaching useful unseen programs. To date, neural program synthesis has been applied to highly simplified programming languages and has used uniform random sampling of the program space to generate a training corpus (and approach that scales for simple languages) [1, 14]. We explore the application of the neural synthesis approach to a far more general programming language; given the non-viability of random sampling in the resulting search space for this language, we propose a novel solution to the corpus generation problem for training, by using weighted genetic sampling combined with iterative automatic refactoring of the neural network's own training corpus based on its self-assessed success.

3 Methodology

In this section we describe the overall architecture of our system. This involves first generating a training corpus, using a synthesis system similar to a genetic algorithm, which uses a fitness function to select parents to reproduce with mutation. This corpus is then used to train a neural network, using the program's behaviour (its I/O mappings) as features, and source code as output labels. The neural network is then able to recognise seen algorithmic behaviour and return source code which can reproduce that behaviour. Rather than simply read off the highest-ranked program, we select N options for each line, and search through a set of programs, to account for imperfections in the neural network's outputs.

3.1 Simplified Language

We designed a simplified C-like programming language, generated functions of which can easily be converted into Java or C-code with a cross-compiler. We designed this language to allow rapid test/execute cycles when generating a training corpus and then searching through a projected search space given by

our neural network. This is possible because the language is directly interpreted, rather than compiled to disk and then executed, and allows us to run around 23,500 programs per second.

Control	Arithmetic	Array
IF VAR GREATER THAN ZERO	ADD	VAR = ARRAY_INDEX
IF VAR1 EQUAL VAR2	SUBTRACT	ARRAY_INDEX = VAR
ELSE	MULTIPLY	ARRAY_INDEX += VAR
LOOP	DIVIDE	
NO-OP	MODULO	
	LITERAL (1,0,-1)	
	INCREMENT	

Fig. 1. The operators available in our simplified language.

To simplify the design of our neural network, we map our language onto the output neurons using a uniform set of possibilities per line. In detail, we logically imagine that each line of a program can have the same $1,332$ different options, derived from 15 operators (see Figure 1), from variable declaration to addition or a loop header. Once a program has been chosen, we check to see if it is syntactically coherent and automatically correct programs that are not. In C-like programs this creates two main corrections: cases in which there are too many 'closing braces', and cases in which there are too few (an unterminated loop). For the former case we simple replace hanging braces with a no-op. In the latter case we insert a closing brace at the very end of a program for any un-closed control blocks; in addition, any un-closed loops are converted to conditional blocks rather than loops. By taking this approach to neuron behaviour uniformity, the neural network does not itself have to learn special cases which limit what each line can be based on prior lines, which would create a much more complex network structure (and, we speculate, a more difficult learning problem).

As further restrictions for this study, in all of our tests, we use programs 9 lines long, padded with the NO_OP operator. We allow 6 integer variables to be accessed by our programs, of which two are fixed and unable to be written to. All of our tests involve passing a single array and a single standalone integer into the program. The two fixed integer variables are the input integer and the length of the input array. The program then has read and write access to both the input array and a second array used as output. These limits allow a wide range of functionalities, while still imposing limits to maintain the problem within computationally tractable sizes.

3.2 Neural Network and Search Architecture

Our code synthesis architecture combines a neural network, used to derive an ordered ranking of possible options, with a search process which iteratively tries these ranked programs up to a configurable search depth.

For this particular study we assume every program can take two parameters: an integer array of length 8 as the first parameter, and an integer as the second. We also assume that every program returns an integer array of length 8. Every cell in an array can hold a value between -8 and 8, while the integer parameter can hold a value between 1 and 4. Reducing the range of the integer parameter

to only positive non-zero values simplifies the search space, as they can always meaningfully use the parameter to refer to an array index.

While our language is capable of representing much more diverse function specifications and numerical ranges (equivalent to C), we use these restrictions as a first step to simplify the search space and neural network complexity. The crucial extension we are targeting is the ability to use LOOP and IF statements, allowing more complex programs in terms of flow than are possible in other code synthesis approaches. We accept a trade off in terms of program length in return for being able to handle a new class of program.

The neural network is then designed as a standard feed-forward architecture as follows. The input layer uses 1,700 input neurons to take 10 I/O examples concatenated together. The output structure uses 9 layers, one for each potential line of a program; each such layer consists of 1,332 neurons, one for every possible way the respective line could be written (including the possibility of a no-op). Internally we use 8 residual layers, each consisting of two dense layers with a width of 512 and an additive layer skip (shown to improve deep networks [15, 11]), and using the ReLu activation function. Dropout was used on all layers, with a probability to keep of 0.75. We used softmax activation for our output layers, and a crossentropy loss function. Our optimizer was the Tensorflow implementation 'RMSPropOptimizer', with learning rate 10^{-5} and momentum 0.9.

The neural network is trained by (automatically) generating a corpus of example programs; the mechanics of this generation are described in detail in the next section. For each generated program in this corpus we randomly generate 10 input/output examples for that program. During training, our randomly generated I/O examples are fed into the neural network's input layer as 170 integer values (each I/O is being composed of 8 values for the input array, one value for the input integer, and 8 values for the output array, this creates 17 values for one I/O example and thus 170 values in total for 10 I/O examples). We choose to encode integers as 10-bit binary numbers for input to the neural network, which was experimentally shown to perform better than using scalar inputs, and so our network has a total of 1,700 input neurons. The network is trained by back-propagating the corresponding output layer neuron values from the actual source code of the corpus program associated with these I/O examples.

Once training is complete, in the testing phase we supply only the 10 I/O examples for a desired program and we use the neural network's probability distribution over its output layer neurons to create a ranked list of programs to search across, from most to least likely. The highest-confidence program would therefore be generated by selecting the highest activity neuron from each output layer. Each layer mapped to a line in the program being generated, and each neuron mapped to one of the 1,332 valid statements which could appear on that line. The 9 highest-activity neurons, one from each of the 9 output layers, therefore map to 9 statements which then make up the highest-confidence program.

To generate a volume of program space, the N highest ranked neurons are chosen per line, giving N ways that particular line could be written in the sampled program. The search volume would therefore consist of every combination

of these options, i.e., $number\_of\_options\_per\_line^{number\_of\_lines}$. For the experiment in this paper, when not otherwise noted, we used 4 options per line for standard programs, and 6 when searching within the human-useful program set.

3.3 Corpus Generation

In a simple DSL, a training corpus for a neural network could be generated by sampling uniformly at random from the space of all possible programs [2]. For our purposes, however, the search space of our more general-purpose language is far too large for uniform random sampling to be effective. When sampled in this way, the resulting corpus of programs is highly repetitive, each program has a high probability of being made up of only (or mostly) lines of code that have no effect, and very few programs contain condition or loop elements (which feature heavily in human-useful programs).

As an alternative to uniform random sampling we designed an approach which combines genetic programming with a set of abstract search biases and a dissimilarity measure. Our generator starts with a seed program, which is an abstract problem reflecting the kinds of search biases that we need; for example a program that uses a loop and a conditional branch, and which reads all of the input array values once and writes each cell of the output array. Starting from this seed program, the genetic algorithm creates iterative populations of mutations. Within a population, we promote code length and an even distributions of all operators, and we penalise writing to loop iterator variables. Finally, mutated programs are only accepted into a population if their are behaviourally dissimilar to the rest of the population. This similarity is measured by feeding 25 randomly generated inputs to each program, and marking the programs dissimilar if any of their output arrays contain a single different value as a result of the inputs. Programs are also rejected if any program reads from or writes to the same memory address in an array twice, further reducing the search space. To gain good learning coverage of flow control, we seed five separate sub-corpuses to form our overall corpus. The first had 0 flow control operators. The second had 1 loop only. The third had 1 loop and 1 CONDITIONAL_GREATER_THAN_0 operator. The fourth had 1 loop and 1 CONDITIONAL_EQUALITY operator. The fifth had 1 loop, 1 CONDITIONAL_EQUALITY and 1 ELSE operator.

The result of this generation process was a diverse set of $10,000$ functionally distinct programs, split between the 5 sub-corpuses of $2,000$ each. In this work we determine functional similarity by feeding both programs a set 25 randomly generated inputs and checking for any difference. We then split these programs amongst training, testing and validation for the neural network. Training received $8,000$ programs, the other two corpuses received $1,000$ programs each. As a result, each corpus' programs were functionally dissimilar, with no program functionality was replicated between corpuses. Note that none of our set of human-useful programs is involved in training the neural network; all such programs are therefore unseen by the system.

3.4 Automatic Corpus Refactoring

Our corpus generation approach tries to train the neural network with a diverse set of programs that facilitate its ability to synthesise human-useful programs. However, corpus generation itself does not necessarily maximise the neural network's internal generality or its use of available model representation space.

We use a novel approach to enhancing the generality and model efficiency of the neural network, by altering the corpus based on the network's own success rate – an approach we term *automatic corpus refactoring*.

The neural network is first trained using the corpus generated as above. It is then asked to locate every program in the training corpus by being given the set of I/O pairs which should result in the given program being found. Because the neural network outputs a ranked list of potential programs, the actual program match may be 10's or 100's of programs down this ranked list. However, during experiments we observed that a *functionally equivalent program* would often exist earlier in the ranking than the exact-match program in the training corpus.

In corpus refactoring, we test to see if such a functionally equivalent program exists earlier in the ranking, and if so we *replace* the training program with this equivalent version. We then retrain the neural network again (with weights re-initialised) based on this new corpus. We can perform this refactoring iteratively, using a new corpus to again replace programs with earlier-found equivalents, until the performance converges to a maximum. As our results demonstrate, refactoring in this form increases performance not just on the training corpus, but also on the testing corpus and on the number of human-useful programs that were correctly constructed – in other words, by adjusting its own training corpus without actually adding any new information, the system is able to find more programs in total than it previously could.

4 Results

This work investigates the effects of automated corpus generation and modification techniques, in the context of trained code synthesis system.

We firstly examine the system's overall code synthesis performance in its intended normal configuration. This allows us to examine its performance, when attempting to solve a human-defined testing corpus of unseen programs. We examine the effects of our automatic refactoring (AR) technique over a set of iterations, to isolate its performance from the initial success of the corpus generation and neural network training steps. AR allows us to improve the performance of a system by adapting its training corpus in response to its current behaviour.

We then investigate the genetic corpus generation approach in further depth, by performing ablation studies on its 'requirements' and fitness function. Following these two studies, we attempt to shed light on the performance gains produced by the AR technique, by examining the changes it makes the corpus.

We evaluate our approach with a set of 'human-useful' programs that the synthesis system is required to find – which we distinguished from the set of programs that the synthesis system finds during its own automated corpus generation phase. For all of our experiments we used Python 3.6.7, Tensorflow 1.12.0 and JVM openjdk 10.0.2 2018-07-17.

4.1 Program Synthesis

To test general program synthesis capability we ran our end-to-end approach 10 times to gain average results. We do this because there are two sources of stochasticity in our approach: the way in which the corpus generation phase works, which is based on randomised mutations; and the way in which the neural network is initially configured, which uses randomised starting weights prior to training. We run corpus generation, five rounds of automated corpus refactoring, and then present the input/output examples for our set of (previously unseen) human-useful programs to see how many the system can find.

Program name	Without AR	With AR	GP n=60	
Absolute	18.18%	**52.94%**	0.00%	The output array is the absolute value of the elements in the input array
Add one	18.18%	52.94%	**85.00%**	The output array is the value of the elements in the input array plus one
Add param	72.73%	88.24%	**96.67%**	The output array is the value of the elements in the input array plus the parameter integer
All negatives to zero	90.91%	**94.12%**	0.00%	The output array is the maximum of (the value in the input array;zero)
Array length	90.91%	94.12%	**100.00%**	The first value of the output array is the length of the input array, the remaining values are 0
Count param	**90.91%**	88.24%	8.33%	The first value of the output array is the number of times the parameter integer appears in the input array, the remaining values are 0
Iterator up to param	63.64%	41.18%	**100.00%**	The first N values of the output array are set to their index, the rest are zero, where N is the parameter integer
Keep above param	0.00%	0.00%	0.00%	The first N values of the output array are the corresponding values in the input array, the rest are 0, where N is the parameter integer
Max(value, param)	0.00%	**5.88%**	0.00%	The output array's values are the maximum of either (the input array's corresponding value;the input parameter)
Min(value,param)	0.00%	**5.88%**	0.00%	The output array's values are the minimum of either (the input array's corresponding value;the input parameter)
Modulo 2	0.00%	0.00%	0.00%	The output array's values are the modulo 2 of the values in the input array
Modulo param	90.91%	**94.12%**	21.67%	The output array's values are the modulo parameter of the values in the input array
Multiply by param	90.91%	**94.12%**	0.00%	The output array is the input array, but offset by N rightwards, losing the last value and setting the first value to zero, where N is the input parameter
Negative	54.55%	**94.12%**	1.67%	The output array's values are the negative values of the values in the input array
Offset by one	27.27%	11.76%	**66.67%**	The output array is the input array, but offset by one rightwards, losing the last value and setting the first value to zero
Offset by param	0.00%	0.00%	0.00%	The output array is the input array offset by N rightwards, losing the last value and setting the first value to zero, where N is the parameter
Return max	0.00%	0.00%	0.00%	The output array's first value is the highest value in the input array, the other values are 0
Return min	0.00%	0.00%	0.00%	The output array's first value is the lowest value in the input array, the other values are 0
Reverse	0.00%	0.00%	0.00%	The output array is the reverse of the input array
Sum values	54.55%	**76.47%**	26.67%	The output array's first value is the sum of all the values in the input array, the other values are zero
Average	38.18%	**44.71%**	25.33%	

Fig. 2. Percentage find rates for two experiment sets, with and without the automated corpus refactoring stage (the first set averaged over 11, and the second over 17 runs). A simple genetic programming algorithm, using the same linguistic constraints, is used as baseline. It can be seen that GP succeeds on simpler problems, but has lower performance when a conditional statement is required.

The results are shown in Fig. 2, detailing both the find rate before any corpus refactoring and also the find rate after the final round of refactoring. For each successfully found program, the neural network has output source code which correctly derives the output from each corresponding input. As an example, for the program "max(value,param)", the array could be [-5,3,-2,3,-4,1,5,8], the parameter '1', and the output [1,3,1,3,1,1,5,8]. From these results we can see that our system locates an average of 38% of our human-useful programs as a result of its initial corpus generation process; this rises to an average of 44% (and a maximum of 60%) after five rounds of corpus refactoring. If we examine

individual programs in this target set, we see that the majority of find rates tend to increase, while a couple of find rates (for example the offset-by-one program) notably decrease. We speculate that the decreases in some program find rates may be caused by those programs lying outside the generalised space into which the neural network moves during corpus refactoring.

We next examine the find-rate of the training, test and human-useful program set over each iteration of automatic corpus refactoring. These results are shown in Fig. 3 for the training and testing sets, and in Fig. 4 for the human-useful set.

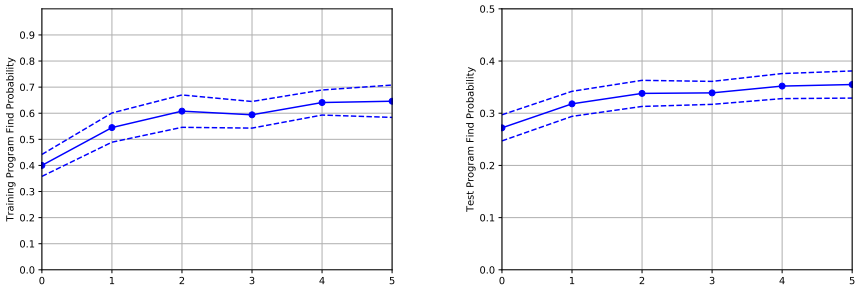

Fig. 3. Success rate on training (left) and test (right) corpus, over each iteration of automatic refactoring, starting from the unmodified corpus, with Standard Deviation

Both the training and test data set show a steady increase in the find-rate of programs from the respective set. For the training set, which shows a find-rate increase from 0.4 to 0.65 (where a value of 1.0 would be all programs found), the process of self-adjusting the training corpus in automatic refactoring clearly shows an enhanced ability to correctly locate more entries in the training corpus. The effect in the test corpus is similar, in this case showing an increase from 0.27 up to 0.36. However, in the case of the test data set the result is much more significant. The increase in find-rate here (i.e., for programs which the system has never seen before) indicates an unexpected phenomena: by having the neural network's training corpus refactored, without adding any data, this allows the neural network to locate more unseen programs than it previously could. It is worth noting that performance decreased in some cases. This is potentially due to the neural network specialising to a particular form of program (the most common) at the expense of others. While this specialisation is overall beneficial, some degradation occurs in certain types of program. We will explore ways to mitigate this effect in future, potentially using a mixture of experts approach employing a set of trained neural networks specialised in different areas.

We see a similar effect in the human-useful programs over successive refactoring iterations, as shown in Fig. 4. Again, all of these programs are unseen by our system during training, but reshaping the training data enables more of them to be successfully synthesised. This suggests that the use of our automated refactoring approach perhaps causes the neural network to become more generalised in its capabilities. However, the data in Fig. 2 provides a more mixed picture: here we see that find-rates for most programs increase after refactoring, but some find rates actually decrease. We explore this further in the next section.

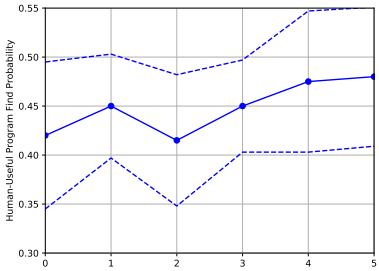

Fig. 4. Success rate on human-useful corpus, over each iteration of the automatic refactoring process, with unchanged corpus as first data point. Corpus size is 20, so each increment of 0.05 corresponds to an average of one program found. 1st Standard Deviation displayed

4.2 Requirements in corpus generation

We now examine the effects of the initial requirements on corpus generation. These requirements are used as input to the genetic algorithm to guide its generation of a set of programs on which the neural network is trained. As a consequence of this training, the neural network is then able to find (or not find) a set of previously unseen human-useful programs. The precise nature of these requirements for corpus generation are therefore an important, if indirect, element of how successful our synthesis approach is at finding programs after training.

In this section we examine how the use of different requirements affects synthesis success. Our complete set of requirements, used across all of the experiments reported so far, includes three major categories as follows.

Array Access This requirement is that all programs containing a loop operator must access every element of the input array. This requirement was included to overcome a perceived problem in the input-access of generated programs. These would often access their inputs in ways which human-written programs rarely would, such as only reading a single element of the input array, or altering their loop iterator and as such skipping elements.

Program Flow This requirement involved subdividing the corpus into 5 subcorpuses, each with its own requirement as to how the flow-control operators should be used. The first corpus required all its programs to have no flow-control operators at all. The second corpus required only a single loop operator. The third required a single loop operator and the first type of conditional operator. The fourth required a single loop operator and the second type of conditional operator. The fifth type required a single loop, the first type of conditional operator and an else block. For each of these corpuses, a single "seed" program was supplied. This program was what we considered to be the "maximally simple" implementation of the requirements; as an example of this the loop-only requirement, from the second corpus, would read in all input values, then write them out unchanged to the output array. The seed programs were implemented due to the genetic search's inability to start generation without them.

Genetic Fitness Function Lastly, our genetic algorithm fitness function rewards particular operator ratios: all operators are expected to be used at least once, with flow control operators in particular weighted twice as highly as others. This was done to promote the use of flow-control, while penalising operators repetition. This was necessary to move away from the "maximally simple" seed programs to those with more commonly useful features.

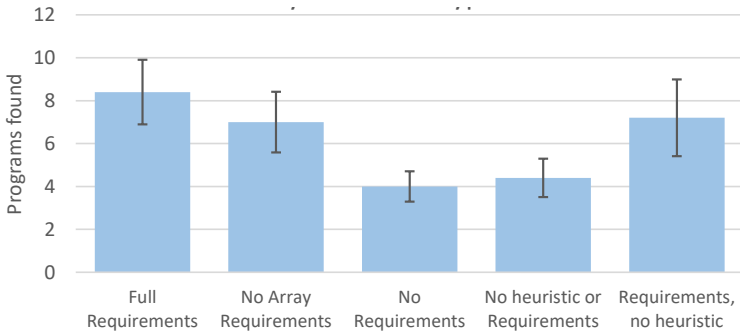

Fig. 5. Average performance for sets of corpuses with varying requirements for constituent generated programs.

We examine the effects of the above requirements by selectively switching them off during corpus generation and comparing how many of our human-useful program set is found as a result. The results are shown in Fig. 5, in which each experiment was run 5 times and we average the data.

The first test shows our full set of requirements, as used in the earlier experiments reported in this section. This achieves the highest performance of any experiment, finding an average of 8.4 programs ($\sigma = 1.5$) from our human-useful target set. The second experiment removes the array access requirement, but keeps the program flow and fitness function heuristics. This performs slightly worse, achieving an average of 7 human-useful programs ($\sigma = 1.4$), indicating that most of our programs are in an area of the total search space in which the input array is uniformly accessed. In the third experiment we removed both the array access requirement and the program flow corpus generation technique, leaving only the fitness function heuristics. This resulted in the worst overall success, with only an average of 4 programs found from our set ($\sigma = 0.71$).

We then removed all requirements and the fitness function heuristics, which actually shows a slight increase in performance with an average of 4.4 ($\sigma = 0.98$) programs found. Finally, we experiment with only using the array access and program flow requirements but remove the fitness function heuristics, which results in the second-best performance overall – indicating that these requirements are more important to success than the fitness function heuristics.

Altogether, these results support the hypothesis that achieving good performance on the human-useful corpus requires a set of corpus generation biases that

are reflective, at a very abstract level, of the typical form of useful programs. The way in which these abstract requirements are communicated to a synthesis system in a human-natural way is a key area of future work.

4.3 Effects of Corpus Refactoring

In this section we examine the effects of automated corpus refactoring in more detail, to better understand why it enables more programs to be found without adding any new data to the system. We characterise this as not adding new data because, even though the training corpus is modified, it is modified only as a result of the neural network's own output from the initial training corpus; the only thing being 'added' is therefore the neural network's apparent preference for which precise form of a target training program to use, but this preference is itself entirely derived from the original training corpus and the neural network's inherent behaviour. We therefore attempt to better understand why this effect occurs, in so far as is possible with the black-box nature of neural networks.

In broad terms, the use of feeding output of one neural network as training labels to another has been demonstrated previously in teacher-student distillation network training [9]; in our case however we believe the success is in fact due to an interplay between the network and the search process. We analyse this effect using the entropy shown by output layers before and after automatic refactoring. These experiments help to verify whether or not the neural network is 'self-generalising' as a result of its search process, or if in fact it is specialising to certain kinds of program in which it tends to become an expert.

In our experiments we measure the entropy of each output layer of our neural network, where each layer corresponds to one line of code. As discussed in Sec. 3.2, each output layer of the network can select from one of a fixed set of possible operations for that line of code – where each option is represented by one neuron. The highest-activated neuron in an output layer is taken as the network's best guess for this line of code. We hypothesise that one of the reasons for corpus refactoring finding extra programs is that the network becomes better generalised in its representation of algorithms. We test this theory by examining the entropy of each output layer – in other words, across all programs, how 'specialised' is each output layer to always choosing the same operation for their line of code, versus their ability to represent a balanced spread of output options. The more balanced the spread is for each output layer, without losing the ability to synthesise programs, the more generalised the neural network may be.

We use the Theil inequality metric to measure the 'inequality' of each line. Maximum inequality (only one option ever used) would be minimum entropy (perfectly predictable). To compute this we find the probability of every option for every line being chosen, across all the programs in the training corpus. If every option were chosen with equal probability, they would each be the average, μ, which is equivalent to $1/N$, where N is the number of options per line.

Fig. 6 shows the results of this experiment. On the left we see the entropy of each output layer before any refactoring has taken place, and on the right we see entropy after five refactoring iterations. We see a clear trend towards a

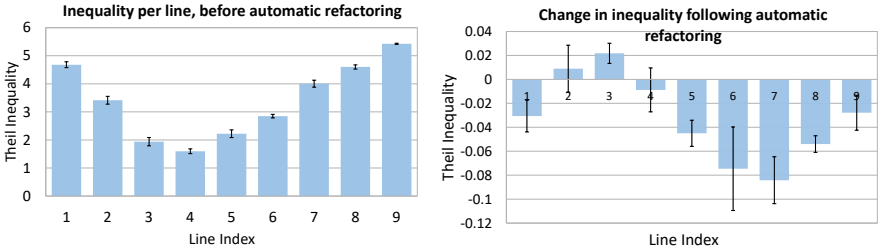

Fig. 6. Reverse entropy of operator distributions by line, as measured by the Theil index. Lower values imply a more even distribution of operator use for a particular, therefore higher entropy.

more balanced ability for each output layer to select a broader range of options, supporting our hypothesis that our refactoring process may aid in generalising the capacity of the neural network.

We can also infer from these experiments that program length becomes more consistent after refactoring. This is to say, if we count the number of non-empty lines (which can appear on any line post-refactoring), we tend towards a closer average across all programs (the length goes from 7.11 lines with a standard deviation of 1.82, to a line average of 7.78 with a standard deviation of 1.41). This suggests that the refactoring process tends to choose longer forms of programs to effectively specialise the network towards programs of a certain length. This is an unexpected duality: as a result of corpus refactoring our network seems to train towards specialising to a certain length of program, which simultaneously generalising itself within that program length by increasing the ability for different lines to take more diverse operations.

5 Conclusion

We have presented an investigation into combining automated corpus generation using a genetic algorithm, with a neural network search technique, to synthesise code in a simplified general-purpose language when given a set of I/O challenges describing the intention of the program.

Our corpus generation is based on a set of highly abstract requirements which align the set of self-generated training programs with roughly the features found in human-useful programs. Our neural network then automatically refines this corpus based on measures of its own success by locating alternative implementations of each program which proved to be higher-ranked in the neural network's own prediction. Together, this technique is able to locate up to 60% of our human useful target programs (which include the synthesis of looping and conditional branch statements) – none of which appeared in the training data.

Our future work will proceed in two main directions: firstly to further explore how initial corpus generation can be easily directed by non-experts when a par-

ticular I/O example cannot be synthesised; and secondly to expand the range of program types that we can synthesise to include those that feature function composition and object instantiation/use. We also intend to further investigate the properties of the neural network itself, to explore how well it generalises in its recognition capabilities to I/O examples of different lengths to those in training, and how the effects of automatic corpus refactoring may be further exploited.

Acknowledgements

This work was supported by the Leverhulme Trust Research Project Grant *The Emergent Data Centre*, RPG-2017-166.

References

1. Balog, M., *et al*: Deepcoder: Learning To Write Programs. ICLR (2017)
2. Chen, X., *et al*: Towards Synthesizing Complex Programs from Input-Output Examples. ICLR pp. 1–31 (2017)
3. Dabhi, V.K., Chaudhary, S.: Empirical modeling using genetic programming: a survey of issues and approaches. Natural Computing **14**(2), 303–330 (2015)
4. Feng, Y., *et al*: Program Synthesis using Conflict-Driven Learning pp. 420–435 (2017)
5. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015 pp. 229–239 (2015)
6. Graves, A., *et al*: Neural Turing Machines. CoRR pp. 1–26 (2014)
7. Graves, A., *et al.*: Hybrid computing using a neural network with dynamic external memory. Nature **538**(7626), 471–476 (2016)
8. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM SIGPLAN Notices **46**(1), 317 (2011)
9. Hinton, G., Vinyals, O., Dean, J.: Distilling the Knowledge in a Neural Network pp. 1–9 (2015), http://arxiv.org/abs/1503.02531
10. Kaiser, Ł., Sutskever, I.: Neural GPUs Learn Algorithms. ICLR pp. 1–9 (2015)
11. Kawaguchi, K., Bengio, Y.: Depth with Nonlinearity Creates No Bad Local Minima in ResNets pp. 1–14 (2018)
12. Petke, J., *et al*: Genetic Improvement of Software: A Comprehensive Survey. IEEE Transactions on Evolutionary Computation **22**(3), 415–432 (2018)
13. Renzullo, J., *et al*: Neutrality and Epistasis in Program Space. Gi pp. 1–8 (2018)
14. Vijayakumar, A., *et al*: Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. ICLR (2018)
15. Wu, S., Zhong, S., Liu, Y.: Deep residual learning for image steganalysis. Multimedia Tools and Applications pp. 1–17 (2017)
16. Yampolskiy, R.V.: Why We Do Not Evolve Software? Analysis of Evolutionary Algorithms. Evolutionary bioinformatics online **14**(1) (2018)