Program Synthesis in a Continuous Space Using Grammars and Variational Autoencoders

David Lynch[1(✉)], James McDermott[2(✉)], and Michael O'Neill[1(✉)]

[1] Natural Computing Research and Applications Group, UCD, Dublin, Ireland
{david.lynch,m.oneill}@ucd.ie
[2] School of Computer Science, National University of Ireland, Galway, Ireland
james.mcdermott@nuigalway.ie

Abstract. An important but elusive goal of computer scientists is the automatic creation of computer programs given only input and output examples. We present a novel approach to program synthesis based on the combination of grammars, generative neural models, and evolutionary algorithms. Programs are described by sequences of productions sampled from a Backus-Naur form grammar. A sequence-to-sequence Variational Autoencoder (VAE) is trained to embed randomly sampled programs in a continuous space – the VAE's encoder maps a sequence of productions (a program) to a point z in the latent space, and the VAE's decoder reconstructs the program given z. After the VAE has converged, we can engage the decoder as a generative model that maps locations in the latent space to executable programs. Hence, an Evolutionary Algorithm can be employed to search for a vector z (and its corresponding program) that solves the synthesis task. Experiments on the program synthesis benchmark suite suggest that the proposed approach is competitive with tree-based GP and PushGP. Crucially, code can be synthesised in any programming language.

1 Introduction

The automatic generation of computer programs has been a goal of researchers in the field of computer science since the origins of the discipline [34]. There are reports of primitive program synthesis in the literature dating back to the 1950's [11,12] with many examples since [13,30,36,37]. The arrival of Genetic Programming and its variants in the late 1980's renewed hopes that programs could be automatically generated by computers. In recent years, researchers in the wider machine learning community have also started to focus on program synthesis [1,14,16,22]. This interest is driven by the expectation that real-world applications of automated program synthesis will have enormous economic and social impact, and will also have important implications for artificial general intelligence [4].

The ability to automatically synthesise programs that solve challenging real-world problems remains an elusive goal. Reasons include the discrete and variable-length nature of computer programs, the non-local mapping between

© Springer Nature Switzerland AG 2020
T. Bäck et al. (Eds.): PPSN 2020, LNCS 12270, pp. 33–47, 2020.
https://doi.org/10.1007/978-3-030-58115-2_3

syntax and semantics, the "all or nothing" aspect of program correctness, and the vast search space.

Genetic Programming (GP) [2,25,35], and its grammar-based variants [29,33], is a form of evolutionary computation [5,21] which can be used for program synthesis. GP routinely achieves human-competitive performance [24] in diverse domains including symbolic regression, architecture, and network optimisation. However, GP has yet to realise its full potential as an engine for automatic programming.

Recently, GP researchers have been calling for an increased focus on program synthesis that embraces techniques drawn from the wider fields of analytics and machine learning [32,34]. One promising research direction looks to combine grammars with autoencoders [20,23]. For instance, Kusner et al. [28] used a combination of grammars and a Variational Autoencoder (VAE) [23] to learn representations for two domains: symbolic regression and drug discovery. The VAE discovers a latent-space encoding of the neighbourhood of sentences in the language expressed by the grammar. The learned representation has appealing properties: it is continuous, approximately normally-distributed, and relatively syntactically and semantically smooth. Thus, powerful numerical optimisation algorithms can be employed to search for new arithmetic expressions or drug molecules.

In this study, we adopt a VAE with a sequence-to-sequence structure, in conjunction with grammars that represent a subset of the Python programming language. The VAE discovers a latent-space encoding of Python programs. An Evolutionary Algorithm [5] is used to successfully search this representation for novel programs. We examine a subset of problems with a range of difficulty drawn from the program synthesis benchmark suite [18].

The remainder of this paper is organised as follows. The grammars, VAE, and evolutionary algorithms are developed in Sect. 2. Our experimental set-up is described in Sect. 3. The proposed approach is benchmarked against canonical grammar-based GP and PushGP [39] in Sect. 4. Finally, we draw conclusions and outline how the algorithms could be improved in Sect. 5.

2 Methods

The main components of our approach are described in this section. Grammars that enable the creation of Python code to solve arbitrary program synthesis tasks are outlined. Our goal is to learn a latent representation of programs using a Variational Autoencoder (VAE). The VAE architecture is presented, and an Evolutionary Algorithm (EA) is developed to search its latent space.

2.1 Grammar Design Pattern

Grammars have commonly been used to represent program spaces in GP [29,33]. However, bespoke grammars had to be written for different program synthesis tasks. Forstenlechner et al. introduced a grammar design pattern to address this inflexibility [9,10]. Their idea is to create a separate grammar for the Boolean,

float, integer, and string data types (see [8]). These sub-grammars are combined depending on what data types are required to solve a problem. An additional grammar constrains the control flow, such as the arrangement of conditionals and loops. Grammars guarantee type safety and they ensure that all individuals are syntactically correct. Runtime exceptions are further reduced via protected methods. Crucially, code can be created in any programming language including Python, Java, C, etc. We synthesise Python code in this paper.

2.2 Variational Autoencoder

VAEs are generative neural models consisting of an encoder and a decoder. In this section, we present an encoder that embeds discrete programs as vectors z in a continuous latent space. The decoder maps points in this latent space back to programs. Our goal is to search for fit programs by performing numerical optimisation in the latent space.

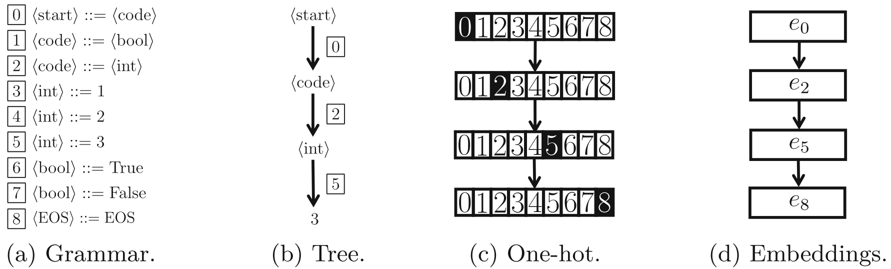


Fig. 1. A toy grammar is displayed in Plot (a). The derivation tree in Plot (b) is realised by expanding production rules 0, 2, and 5. In Plot (c), the program is given by a sequence of one-hot vectors (including an 'EOS' token). Finally, each one-hot vector is associated with a learned embedding vector in Plot (d).

Initially, a grammar for the program synthesis task is formed as outlined in Sect. 2.1. The VAE is trained on a corpus of programs sampled from this grammar. For example, consider the toy grammar and the sampled program displayed in Fig. 1. The production rules used to generate the program are represented by a sequence of one-hot vectors $\tilde{o} = (o_0, o_2, o_5, o_8)$. The associated embeddings $\tilde{e} = (e_0, e_2, e_5, e_8)$ are then passed as inputs to the encoder.

We adopt a bidirectional [38] gated recurrent unit network [3] (BiGRU) as the encoder. A recurrent model lends itself naturally to modelling sequences of production rules. Furthermore, the BiGRU can deal with input sequences of an arbitrary length (that is, programs of different sizes). The flow of information through the encoder is illustrated in Fig. 2. A sequence of embeddings[1] $\tilde{e}$ is provided as an input to the BiGRU. The function $\overrightarrow{G}(\cdot)$ is a GRU cell [3] that

[1] Only (e_2, e_5) are displayed for clarity, but in practice (e_0, e_2, e_5, e_8) would be used.

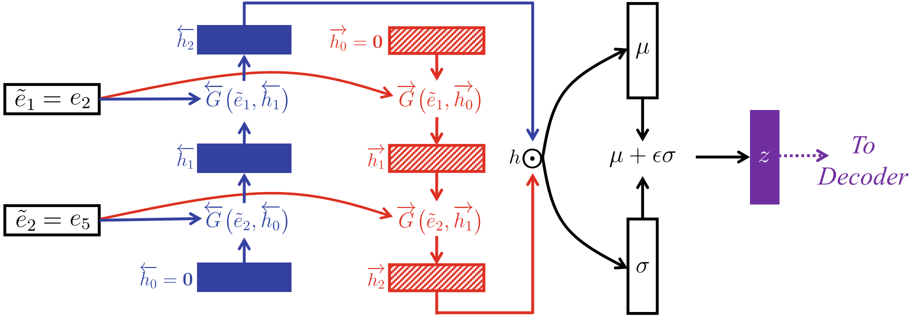


Fig. 2. The encoder maps embeddings $\tilde{e}$ of the production rules used to generate a program to a latent representation z.

integrates the current embedding vector $\tilde{e}_t$ with the previous forward hidden state $\overrightarrow{h}_{t-1}$ to give $\overrightarrow{h_t}$. Similarly, $\overleftarrow{G}(\cdot)$ updates the previous backward hidden state. The hidden states emerging from the BiGRU are concatenated to yield a summary of the program h. Hence, the latent code z is given by:

$$z = (W_{h\mu}h + b_\mu) + \epsilon\,(W_{h\sigma}h + b_\sigma) = \mu + \epsilon\sigma, \tag{1}$$

where the weight matrices W and bias vectors b are learned parameters of the model. Variables μ and σ are interpreted as the mean and standard deviation of a multivariate normal distribution $\mathcal{N}(\mu, \sigma)$, and ϵ is sampled from the multivariate standard normal distribution $\mathcal{N}(0, 1)$. The auxiliary variable ϵ allows gradients to flow backwards through the network [23].

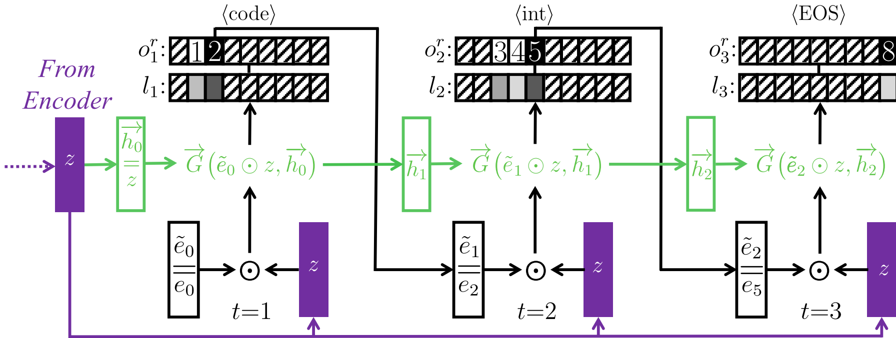


Fig. 3. The decoder reconstructs the sequence of one-hot vectors $\tilde{o}$ that encode a program given its latent representation z.

The decoder in Fig. 3 reconstructs the program given z. We implement the decoder as a forward GRU network. The outputs are one-hot vectors o^r, which indicate the predicted production rule at a given timestep. Two inputs are

provided to the GRU cell $\overrightarrow{G}(\cdot)$ at each timestep. The first input is the previous hidden state (initialised to $\overrightarrow{h_0} = z$). The second input is the vector $\tilde{e}_{t-1}$ concatenated with z, where $\tilde{e}_{t-1}$ is the embedding of the production rule from the previous timestep (initialised to $\tilde{e}_0 = e_0$). Utilising an autoregressive input helps the decoder keep track of previously selected production rules.

Recall that the program in Fig. 2b is described by one-hot vectors $\tilde{o} = (o_0, o_2, o_5, o_8)$, mediating the expansion $\langle \text{start} \rangle \rightarrow \langle \text{code} \rangle \rightarrow \langle \text{int} \rangle \rightarrow 3$ plus a final 'EOS' token. A sequence of reconstructed one-hot vectors o^r are computed by the decoder as follows:

- $t = 0$: we can set $o^r_0 = [1, 0, 0, 0, 0, 0, 0, 0, 0]$ since all derivation trees have $\langle \text{start} \rangle$ at their root node.
- $t = 1$: the non-terminal to be expanded at $t = 1$ is $\langle \text{code} \rangle$. In order to select a production rule, the GRU cell emits a hidden state h_1 from which the logit $l_1 = \mathit{softmax}(\mathit{mask}(W_{hl} h_1 + b_l))$ is computed. The logit defines a probability distribution over production rules – a mask is applied because $\langle \text{code} \rangle$ can only be expanded using rules 1 or 2 (see Fig. 2a). Since the maximum value of l_1 occurs at index 2, it follows that rule 2 is selected and $o^r_1 = [0, 0, 1, 0, 0, 0, 0, 0, 0]$.
- $t = 2$: the $\langle \text{int} \rangle$ non-terminal can be expanded using rules 3, 4, or 5. Rule 5 is selected giving $o^r_2 = [0, 0, 0, 0, 0, 1, 0, 0, 0]$.
- $t = 3$: finally the 'EOS' token is reached indicating that the decoding process has terminated, and $o^r_3 = [0, 0, 0, 0, 0, 0, 0, 0, 1]$.

At test time, we disregard the encoder and engage the decoder as a generative model to search for fit programs. A program is constructed by passing a point z in the latent space to the decoder. Non-terminals are expanded in a depth-first manner using the one-hot vectors (that is, production rule choices) produced by the decoder.

VAE Loss Function: In summary, a program is described by a sequence of one-hot vectors (production rules) $\tilde{o}$ and their corresponding embeddings $\tilde{e}$. We propose to learn a continuous latent representation of programs using a VAE. The encoder $q_\phi(z|\tilde{e})$ maps $\tilde{e}$ to a latent code z. The decoder $p_\theta(\tilde{o}|z)$ is a generative model that reconstructs $\tilde{o}$ given z. Parameters ϕ and θ are jointly optimised via gradient descent on the loss function:

$$\mathcal{L}\left(\phi, \theta; \tilde{o}, \tilde{l}, z\right) = \mathcal{L}_{\text{AE}}\left(\tilde{o}, \tilde{l}\right) + \mathcal{L}_{\text{REG}}(z), \tag{2}$$

where $\mathcal{L}_{\text{AE}}\left(\tilde{o}, \tilde{l}\right)$ denotes the reconstruction loss, and $\mathcal{L}_{\text{REG}}(z)$ is a regularisation loss encouraging latent codes to be normally distributed.

The reconstruction loss is defined as the cross entropy between $\tilde{o}$ and the logits $\tilde{l}$ (see Fig. 3) emitted by the decoder:

$$\mathcal{L}_{\text{AE}}\left(\tilde{o}, \tilde{l}\right) = -\frac{1}{|\tilde{o}|} \sum_{t=1}^{|\tilde{o}|} \tilde{o}_t \cdot \log_e\left(\tilde{l}_t\right).$$

To shape the latent space, we adopt the maximum-mean discrepancy (MMD) regularisation loss proposed by Zhao et al. [41]:

$$\mathcal{L}_{\text{REG}}(z) = \text{MMD}(z, z'),$$

where $z \sim q_\phi(z|\tilde{o})$ is the latent vector produced by the encoder, and z' is sampled from a multivariate standard normal distribution. The latent space realised by minimising Eq. 2 should exhibit two properties that enable effective search. Firstly, programs should be densely distributed near the origin. Secondly, nearby points should decode to syntactically similar programs (high syntactic locality).

2.3 Evolutionary Algorithms

Evolutionary Algorithm: An Evolutionary Algorithm (EA) is implemented to search the real-valued space discovered by the VAE. Programs are represented by real vectors considered as locations in the learned representation. Initialisation, mutation, and crossover are defined on real vectors. An initial population is obtained by sampling 1000 individuals z from a standard normal distribution. This initial population is evolved over 300 generations as follows.

In every generation, individuals are assigned a fitness by decoding z to give a derivation tree (program), which is then evaluated on the program synthesis task. Fit programs are selected using tournament selection or lexicase selection [19]. Selected individuals undergo mutation and crossover. An individual z is mutated by adding to it a vector Δz sampled from a standard normal. Crossover is applied to every pair of selected individuals. Elements are marked for crossover with probability 0.1. Hence, marked elements m in parents p^1 and p^2 are interpolated to yield children c^1 and c^2 such that:

$$c^1_m = p^1_m + i_1 \times (p^2_m - p^1_m),$$
$$c^2_m = p^2_m + i_2 \times (p^1_m - p^2_m),$$

where i_1 and i_2 are drawn from a uniform distribution $\mathcal{U}(0, 1)$. Seven elites enter the next generation without undergoing crossover or mutation.

Hill Climbing: The EA is benchmarked against a hill climbing algorithm in order to assess the need for population-based search. Here, a single individual z_{best} is initialised and evaluated. A new hypothesis z_{hyp} is generated by adding a sample from a standard normal Δz to z_{best}. Hypotheses are evaluated over $pop\ size \times gens = 1000 \times 300$ iterations. After every iteration, z_{hyp} replaces z_{best} if the corresponding program attains a better fitness.

Genetic Programming: The EA and hill climbing algorithm perform search in a continuous latent space. By contrast, Grammar-based Genetic Programming (GP) [7,29] explores the discrete space of derivation trees directly. An initial population of 1000 randomly generated derivation trees is formed using the ramped

half-and-half method. In every generation, individuals which are selected using tournament selection undergo subtree mutation and crossover. Subtree mutation replaces a randomly selected subtree with a new randomly generated subtree. Subtree crossover swaps randomly selected subtrees (with the same root node) between two parents. Every individual undergoes mutation, and the crossover probability is 0.9. We use generational replacement with elitism (the elite size is 7). Derivation trees are allowed to grow to a maximum depth of 16.

3 Experimental Setup

Within the GP community, a program synthesis benchmark suite has been proposed [18], composed of 29 problems which might typically be assigned as exercises to beginner programming students. The problems are all specified as word problems, with recommendations for generating correct input/output pairs and a train/test split. They require the use of multiple data types and control structures including loops. Recent work on program synthesis has made good progress on this suite [9,10,17,19]. In this proof of concept study we examined six problems of varying difficulty drawn from the suite: `grade`, `last_index_of_zero`, `median`, `negative_to_zero`, `smallest`, and `vectors_summed`.

For each problem, VAEs were trained using training and development sets containing 49000 and 1000 programs respectively. Programs were sequences of production rules (encoded as one-hot vectors) sampled from a grammar. The grammars (one per problem) were assembled based on the data types required to solve a problem (see Sect. 2.1 and in [8]). The best VAE from ten independent runs was combined with the EA from Sect. 2.3 to enable program discovery. The hyperparameters displayed in Table 1 were determined by trial-and-error.

Table 1. VAE hyperparameter settings.

Epochs	100
Initial learning rate	0.01
Learning rate decay rate (per epoch)	0.95
Batch size	128
Dimensionality of the latent space	50
Dimensionality of the hidden states	50
Dimensionality of the embeddings	50
Optimisation algorithm	RMSprop
Gradient clipping	Norm of gradients $\leq$ 0.00001
Model selection based on	Development set loss

The evolutionary algorithms outlined in Sect. 2.3 were deployed on six problems drawn from the benchmark suite. For a given problem, 100 independent runs of the EA-VAE and GP algorithms were carried out. Tournament selection was adopted in one set of runs, and lexicase selection was used in another set. Similarly, 100 runs of the hill climbing algorithm were executed for each problem.

4 Results and Discussion

We compare the proposed algorithms for automatically synthesising Python code in this section. Success rates are reported on the training and test sets of six problems drawn from the benchmark suite. We illustrate how transitions between neighbouring points in the latent space map to smooth syntactic transitions in program space. Finally, analysis of the fitness landscape reveals why some problems are harder for the EA-VAE to solve than others.

4.1 Success Rates

The EA-VAE and GP success rates are displayed in Table 2. Both algorithms solve more problems when lexicase selection is used to select parents; GP solves all six problems, while the EA-VAE discovers solutions for every problem except `grade`. Neither algorithm solves `vectors_summed` under tournament selection, but they both find solutions under lexicase selection.

Table 2. The reported results include: the success rates (out of 100 runs) on training and test sets, median number of production rules consumed when generating programs, and the median generation at which solutions were discovered. Results are given under tournament selection and lexicase selection.

Problem	EA-VAE				GP			
	Train	Test	Rules	Gen	Train	Test	Rules	Gen
`grade`	0	0	89	NA	10	**4**	404	204
`median`	97	**97**	22	50	77	27	329	102
`last_index_of_zero`	5	5	28	155	16	**14**	299	104
`negative_to_zero`	83	**83**	18	33	50	47	314	13
`smallest`	100	**100**	22	18	100	86	149	12
`vectors_summed`	0	0	23	NA	0	0	200	NA

(a) Tournament Selection.

Problem	EA-VAE				GP			
	Train	Test	Rules	Gen	Train	Test	Rules	Gen
`grade`	0	0	141	NA	85	**31**	362	97
`median`	100	**100**	22	24	100	49	214	14
`last_index_of_zero`	2	1	46	147	33	**30**	267	91
`negative_to_zero`	64	64	43	69	72	**68**	223	21
`smallest`	100	**100**	23	12	100	89	86	4
`vectors_summed`	7	7	69	99	20	**14**	270	120

(b) Lexicase Selection.

Table 3. The proposed approach is benchmarked against hill climbing (HC-VAE), tree-based GP, and PushGP. Success rates are reported on the test sets of each problem. Lexicase selection was used in the EA-VAE, GP, and PushGP runs. The results for PushGP are taken from [18].

Problem	EA-VAE	HC-VAE	GP	PushGP
`grade`	0	0	**31**	4
`median`	**100**	1	49	45
`last_index_of_zero`	1	0	**30**	21
`negative_to_zero`	64	0	**68**	45
`smallest`	**100**	24	89	81
`vectors_summed`	7	0	**14**	1

Programs evolved by the EA-VAE algorithm typically generalise perfectly from train to test cases. The EA-VAE generalises well because it gives rise to near minimal programs. Comparing the columns labelled "Rules" in Tables 2a and 2b, we see that GP consumes many more production rules than the EA-VAE. That is, GP is more susceptible to bloat. Introns may be beneficial to GP during evolution [31], but their presence impacts generalisation to the test sets.

The success rates displayed in Table 3 confirm that the EA outperforms greedy hill climbing. A fitness landscape analysis will reveal why the latent space is not amenable to greedy search. The EA-VAE achieves the highest success rates on two problems. However, GP is the most consistent algorithm overall, finding multiple solutions to every problem. Unlike PushGP, the grammar-based techniques generate interpretable Python programs, such as those in Fig. 4.

```
in0.insert(i1,i0)
in1.insert(i1,max(i0, i2))
in1.insert(i1,max(i0, i2))
in1.insert(max(i1, i0),getIndexIntList(in0, i2))
in1.insert(getIndexIntList(in0, int(3.0)),getIndexIntList(list(saveRange(i1,i0)), getIndexIntList(res0, i0)))
its = 0\nfor i1 in in1:{:\nin0.insert(max((i0 - i0), getIndexIntList(res0, i0)),min(divInt(i1,i0), abs(i0)))\nif its > 100:{:\nbreak\n:}its += 1\n:}
its = 0\nfor i1 in in1:{:\nres0.append(max((i0 - i0), divInt(i0,i1)))\nif its > 100:{:\nbreak\n:}its += 1\n:}
its = 0\nfor i0 in in1:{:\nres0.append(max((i0 - i1), min(i0, i0)))\nif its > 100:{:\nbreak\n:}its += 1\n:}
its = 0\nfor i0 in in1:{:\nres0.append(max((divInt(i0,i1) + min(i0, i0)), getIndexIntList(in0, len(in0))))\nif its > 100:{:\nbreak\n:}its += 1\n:}
its = 0\nfor i0 in in1:{:\nres0.append((max(divInt(i0,i1), min(i0, i0)) + getIndexIntList(in0, len(res0))))\nif its > 100:{:\nbreak\n:}its += 1\n:}
```

(a) `vectors_summed`

```
b0 = -in0 > i0
b0 = in0 != min(in2, i0)
res0 = min(in0, in2)
res0 = max(in0, min(in2, i0))
res0 = max(min(in0, in2), i0)
res0 = min(max(in0, in2), i0)
res0 = min(max(in0, in1), in2)
res0 = min(max(in0, in1), min(in2, in0))
res0 = min(max(in0, in1), in2)
res0 = min(max(in0, in1), max(in2, min(in0, in1)))
```

(b) `median`

```
i0 = in1
res0 = in1
res0 = in1
res0 = min(in1, in2)
res0 = min(in1, in2)
res0 = min(in2, in1)
res0 = min(in2, mod(in1,i1))
res0 = min(mod(in2,in1), min(i1, res0))
res0 = min(mod(in2,in1), min(i1, in1))
res0 = min(mod(abs(in2),min(in1, i1)), min(min(in1, in0), min(in2, in3)))
```

(c) `smallest`

Fig. 4. We interpolate between a random point in the latent space z_r, and one of the solutions found the EA z_{EA}. Programs are displayed for points $z_r + \delta(z_{EA} - z_r)$, where $\delta \in [0.0, 0.1, \ldots, 1.0]$. Note that the "\ n" symbols indicate line breaks.

The interpolations in Fig. 4 suggest that the VAE learns a relatively smooth and coherent latent space. For example, consider the interpolations for `vectors_summed`. The concept of a for-loop appears, and is retained, as we move closer to the solution z_{EA}. Further refinements of the loop body yield a program (red text) that achieves the desired semantics: it returns a vector 'res0', which is the summation of input vectors 'in0' and 'in1'. Evidence of gradual syntactic transitions implies that the VAE packs programs densely around the origin. This property of the latent space arises due the regularisation term in Eq. 2.

4.2 Landscape Analysis

The Cartesian space allows natural methods of landscape analysis. Figure 5 shows how fitness changes over interpolations between solutions (found by EA) and random points (sampled from a standard normal in the VAE latent space). The fitness landscape is characterised by neutrality and discrete steps in fitness. Nonetheless, there is evidence of a positive *fitness-distance correlation* (FDC).

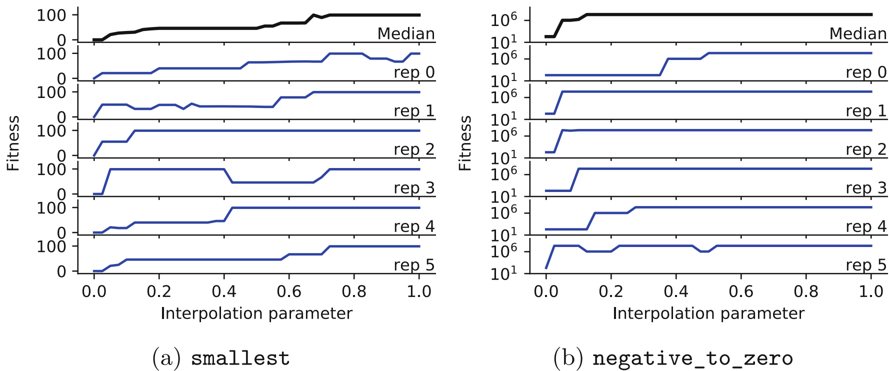
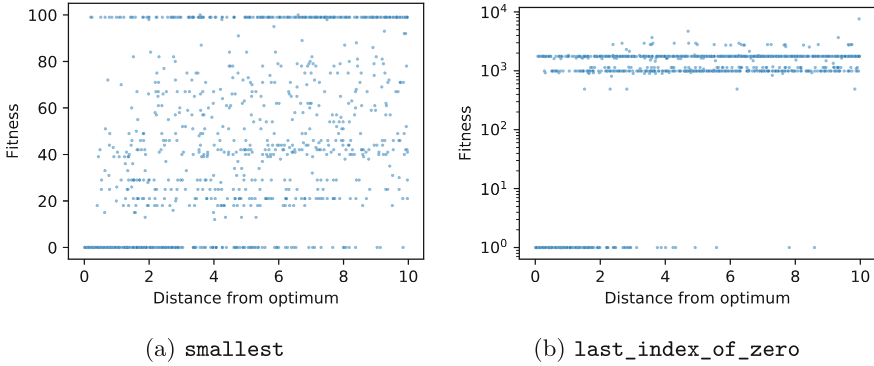


Fig. 5. Fitness over interpolations. We show 6 repeats, and the median over 30.

To expand on this evidence, we also present FDC results where points are sampled rather than created by interpolation. In particular, for each trial we randomly choose a solution z from among those found by the EA, and then sample a random vector y from a standard normal. Because of the high dimension (50), this gives a strong bias for Euclidean distance $5 \leq d(z, y) \leq 8$. A solution is to then scale y to a desired length, and we have chosen to scale y so that $d(z, y)$ is distributed uniformly on $[0, 10]$. The scatter plots in Fig. 6 allow us to see the fitness-distance relationship over the whole space, and also focus on the relationship for small distances.

Figure 6 indicates that several solutions exist in the region around a given solution (where a 'solution' has fitness 0). Therefore, the EA is not confronted with a needle-in-a-haystack fitness landscape. As expected, increasingly fewer solutions are observed as we move further away from a known solution in the

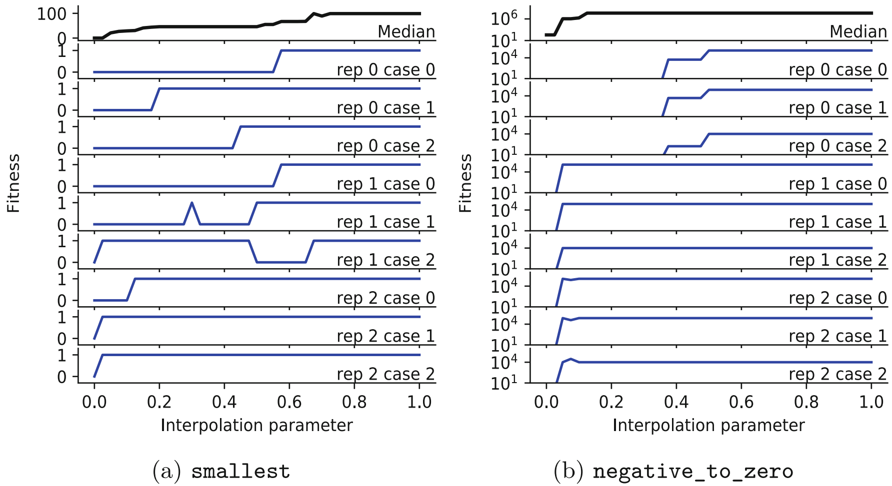


(a) `smallest` (b) `last_index_of_zero`

Fig. 6. Fitness against distance. On the right hand side, we have plotted *fitness* + 1 to allow a log-plot, so 10^0 indicates a solution. A few outliers are excluded.

Table 4. FDC values, where R is Pearson's correlation (excluding outliers), and τ is Kendall's. The `grade` problem is excluded because we found no solutions, and hence cannot compute an FDC value.

Problem	R	τ
`median`	0.30	0.20
`smallest`	0.23	0.16
`negative_to_zero`	0.22	0.15
`vectors_summed`	0.19	0.17
`last_index_of_zero`	0.07	0.06



(a) `smallest` (b) `negative_to_zero`

Fig. 7. Fitness over interpolation: as in Fig. 5, but showing the error on 3 repeats and 3 individual training cases as indexed on the right.

latent space (lower right). Table 4 shows that `last_index_of_zero`, a hard problem for VAE-based search, has FDC near 0, while easier problems show increasingly larger positive FDC values. Thus, FDC partly explains performance.

Because lexicase selection considers errors on individual training cases, it is interesting to consider them separately as in Fig. 7. As expected we see some evidence of correlation among cases.

5 Conclusions and Future Work

Variational Autoencoders (VAEs) are effective at learning a coherent continuous representation of discrete programs. Solutions to non-trivial synthesis problems are discovered by searching the VAE's latent space using an Evolutionary Algorithm (EA). The EA-VAE approach to program synthesis is competitive with tree-based GP and PushGP on problems drawn from the benchmark suite. However, some problems present a neutral and discretised fitness landscape, resulting in lower success rates for the EA-VAE versus the benchmarks.

The algorithm could be improved in a variety of ways. Firstly, it will be interesting to explore techniques for better organising the latent space. One possibility, inspired by Gómez-Bombarelli et al. [15], is to jointly train a multilayer perceptron (MLP) with the VAE. The MLP could be trained to predict program semantics or the program's fitness on test cases, given the VAE's latent layer z as input. This would encourage program semantics information to be present, and well-structured, in the latent layer. Secondly, a more informative fitness function could be used to guide the search algorithm. We used the raw errors on input/output training pairs. However, program synthesis is not truly a black-box problem. There is a wealth of additional information that can be made available to the search algorithm, such as the program execution trace and the semantics on individual inputs [26,27]. Finally, state of the art natural language models, such as the transformer [40] or BERT [6], could be easily incorporated into the VAE's architecture. These ideas can be assessed on the full benchmark suite, and on more recently proposed benchmarks such as the ARC problems [4].

Our approach to program synthesis combines the two dominant paradigms in artificial intelligence: symbolic AI and connectionism. On the one hand, we evolve symbolic programs that can express abstract concepts, generalise perfectly, and that can be interpreted by humans. On the other hand, programs are embedded in the latent space using a neural network. This class of models are adept at pattern recognition, data compression, and representation learning. Discrete search in the space of symbolic programs will be a cornerstone of artificial intelligence research in the coming decades. We believe that hybridising the symbolic and connectionist paradigms is a promising research direction.

Acknowledgements. This research is based upon works supported by the Science Foundation Ireland under grant 13/IA/1850.

References

1. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: learning to write programs. In: Proceedings International Conference on Learning Representations 2017. OpenReviews.net (2017)
2. Orzechowski, P., Magiera, F., Moore, J.H.: Benchmarking manifold learning methods on a large collection of datasets. In: Hu, T., Lourenço, N., Medvet, E., Divina, F. (eds.) EuroGP 2020. LNCS, vol. 12101, pp. 135–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44094-7_9
3. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
4. Chollet, F.: The measure of intelligence. arXiv preprint arXiv:1911.01547 (2019)
5. De Jong, K.A.: Evolutionary Computation: A Unified Approach. MIT Press, Cambridge (2006)
6. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
7. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., O'Neill, M.: PonyGE2: grammatical evolution in python. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1194–1201 (2017)
8. Forstenlechner, S.: Program Synthesis with Grammars and Semantics in Genetic Programming. PhD Thesis pp. 162–175 (2019)
9. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 262–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_17
10. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) PPSN 2018. LNCS, vol. 11101, pp. 197–208. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99253-2_16
11. Friedberg, R.M.: A learning machine: part i. IBM J. Res. Dev. **2**(1), 2–13 (1958)
12. Friedberg, R.M., Dunham, B., North, J.H.: A learning machine: part ii. IBM J. Res. Dev. **3**(3), 282–287 (1959)
13. Fujiki, C., Dickinson, J.: Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. In: Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, July 1987. pp. 236–240 (1987)
14. Gaunt, A.L., et al.: TerpreT: A probabilistic programming language for program induction. CoRR abs/1608.04428 (2016)
15. Gómez-Bombarelli, R.: Automatic chemical design using a data-driven continuous representation of molecules. ACS central science **4**(2), 268–276 (2018)
16. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. SIGPLAN Notices **46**(1), 317–330 (2011)
17. Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 937–944 (2017)
18. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1039–1046 (2015)

19. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE T. Evolut. Comput. **19**(5), 630–643 (2014)
20. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786), 504–507 (2006)
21. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology. Control and Artificial Intelligence. MIT Press, Cambridge (1975)
22. Katayama, S.: Recent Improvements of magichaskeller. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 174–193. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_9
23. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)
24. Koza, J.R.: Human-competitive results produced by genetic programming. Genet. Program. Evol. Mach. **11**(3–4), 251–284 (2010)
25. Koza, J.R., Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, vol. 1. MIT press, Cambridge (1992)
26. Krawiec, K., O'Reilly, U.M.: Behavioral programming: a broader and more detailed take on semantic GP. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pp. 935–942 (2014)
27. Krawiec, K., Swan, J.: Pattern-guided genetic programming. In: Proceedings of the 15th Annual Conference On Genetic And Evolutionary Computation, pp. 949–956 (2013)
28. Kusner, M.J., Paige, B., Hernández-Lobato, J.M.: Grammar variational autoencoder. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. pp. 1945–1954. JMLR. org (2017)
29. Mckay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genet. Program. Evol. Mach. **11**(3–4), 365–396 (2010)
30. Muggleton, S.: Inductive logic programming: issues, results and the challenge of learning language in logic. Artif. Intell. **114**(1–2), 283–296 (1999)
31. Nordin, P., Francone, F., Banzhaf, W.: Explicitly defined introns and destructive crossover in genetic programming. Adv. Genetic Program. **2**, 111–134 (1995)
32. O'Neill, M., Fagan, D.: The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI. GEC, pp. 179–192. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-04735-1_9
33. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language (2003)
34. O'Neill, M., Spector, L.: Automatic programming: The open issue? Genetic Programming and Evolvable Machines pp. 1–12 (2019)
35. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A Field Guide to Genetic Programming. Lulu.com (2008)
36. Rich, C., Waters, R.C.: Automatic programming: Myths and prospects. Computer **21**(8), 40–51 (1988)
37. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM J. Res. Dev. **3**(3), 210–229 (1959)
38. Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE Trans. Signal Process. **45**(11), 2673–2681 (1997)

39. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, pp. 1689–1696 (2005)
40. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)
41. Zhao, S., Song, J., Ermon, S.: InfoVAE: Information maximizing variational autoencoders. arXiv preprint arXiv:1706.02262 (2017)