Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks

Paweł Liskowski
Poznan University of Technology
Poznan, Poland
pliskowski@cs.put.poznan.pl

Iwo Błądek
Poznan University of Technology
Poznan, Poland
ibladek@cs.put.poznan.pl

Krzysztof Krawiec
Poznan University of Technology
Poznan, Poland
krawiec@cs.put.poznan.pl

ABSTRACT

When search operators in genetic programming (GP) insert new instructions into programs, they usually draw them uniformly from the available instruction set. Prefering some instructions to others would require additional domain knowledge, which is typically unavailable. However, it has been recently demonstrated that the likelihoods of instructions' occurrence in a program can be reasonably well estimated from its input-output behavior using a neural network. We exploit this idea to bias the choice of instructions used by search operators in GP. Given a large sample of programs and their input-output behaviors, a neural network is trained to predict the presence of individual instructions. When applied to a new program synthesis task, the network is first queried on the set of examples that define the task, and the obtained probabilities determine the frequencies of using instructions in initialization and mutation operators. This priming leads to significant improvements of the odds of successful synthesis on a range of benchmarks.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**; • **Theory of computation →** *Evolutionary algorithms*; • **Computing methodologies → Neural networks**;

KEYWORDS

program synthesis, genetic programming, neural networks, search prioritization

ACM Reference Format:
Paweł Liskowski, Iwo Błądek, and Krzysztof Krawiec. 2018. Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks . In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205629

1 INTRODUCTION

In program synthesis from examples, a large combinatorial space is searched in order to identify the program that exhibits the mandated input-output behavior. The search can be conducted with a range of algorithms, which – unless naïve – *prioritize* (or *bias*) their policy using some means. In genetic programming (GP), the means of prioritization are the search operators used and the fitness function that measures the 'goodness of fit' of a candidate program to examples. Such prioritization can be characterized as *posterior*, as it is administered *in response* to candidate programs only once they have arisen in search and evaluated, and *local*, as this is being done on individual, per-program basis.[1]

In contrast, in a recent work, Balog et al. [2] proposed DeepCoder, an *a priori* and *global* approach to 'priming' of program synthesis. The authors demonstrated that a neural network can be trained that maps the input-output examples to the *probability distribution* of instructions to be used in the synthesized programs. A search algorithm then follows the guidance of the network (i.e., its response to a given set of input-output examples) and prefers using some instructions to others. When combined with systematic breadth-first search and other search algorithms, DeepCoder significantly reduces the time-to-success compared to unprioritized search, sometimes by orders of magnitude (details in Section 2).

In this study, we propose an approach that engages a similar neural network to bias evolutionary search in GP (Section 3). The trained network is queried on the synthesis task (examples) before a GP run, and the obtained probabilities determine the likelihoods of using instructions in search by priming the search operators (in particular the mutation operator). In this way, we combine the posterior search prioritization typical for GP with the a priori biasing proposed in DeepCoder. When applied to a range of benchmarks, the method proves superior to two baseline approaches, one being the standard, unbiased GP, and the other being GP biased with the probability of incidence of instructions in the training set.

2 DEEPCODER

DeepCoder [2] is predicated on the assumption that useful information about the program to be synthesized can be obtained from the training examples. As this conjecture underpins all methods that synthesize programs from examples, including GP, it is not particularly original in itself. However, DeepCoder attempts to obtain such information *directly* from the examples, without the costly generate-and-test typical for evolutionary approaches, and without an understanding of the semantics of instructions of the programming language. This is achieved by building a neural network that predicts the likelihoods of instructions from the supplied training examples. We proceed with detailing the operation of DeepCoder as it is applied to a given synthesis problem, and only then, in Sections 2.3 and 2.5 present network architecture and the training

[1]Search is also biased by the choice of instruction set, maximum program length and other parameters, but in this paper we assume those aspects to be given and fixed.

```
a ← [int]
b ← [int]
c ← ZipWith (−) b a
d ← Count (>0) c
```

Listing 1: Exemplary program in the considered DSL (P2 from [2])

```
x ← [int]
y ← [int]
c ← Sort x
d ← Sort y
e ← Reverse d
f ← ZipWith (*) d e
g ← Sum f
```

Listing 2: Program P4 from [2].

process. Importantly, this description reflects our reimplementation of DeepCoder, which involved certain design choices (while [2] suggests at places a wider range of variants).

The authors of [2] put DeepCoder in a wider framework of *Learning Inductive Program Synthesis*, comprising specification of a domain-specific programming language (DSL), a machine learning model that maps the examples to probabilities of occurrence of instructions in programs (a neural network), the procedure of generating training data for training that model, and a search procedure. In the following, we cover the two former components, postponing the description of the latter two to the experimental section.

2.1 The domain-specific language

The DSL used in [2] is quite sophisticated in featuring a relatively rich instruction set and allowing for high-level functions. It can be likened to linear GP: a program is a fixed-length sequence of instructions, each of which issues a function call, creates a fresh variable, and assigns the outcome of the former to the latter. The language involves two data types: integers (int) and lists of integers ([int]) (some functions accept also other arguments, but they cannot be stored in variables; see explanations in the following). The set of instructions is typical for list-manipulation functions and comprises Head, Last, Take, Drop, Access, Minimum, Maximum, Reverse, Sort, Sum, and the higher-order functions Map, Filter, Count, ZipWith, ScanL. Importantly, DSL comprises not only the top-level functions that may form the right-hand side of the instructions (like the ZipWith in Listing 1), but also several other lexical elements of the language:

- lambdas for Map (add1, sub1, multMinus1, mult2, mult3, mult4, div2, div3, div4, square).
- predicates for Filter and Count (>0, <0, isOdd, isEven).
- lambdas for ZipWith and ScanL (+, -, *, min, max).

Listing 1 presents an exemplary program that accepts two lists of integers and counts how many times an element in the former one is greater than the corresponding element in the latter (Program 2 from Appendix A in [2]). The first two instructions fetch the input arguments into variables a and b, and as such do not contribute to program length (which is thus 2 in this case). The output of the program is the value of the last assigned variable – in this case an integer, the value of variable d.

Listing 2 presents a more sophisticated program that calculates the minimum total area of rectangles that can be constructed from the sides of lengths provided in two input lists. Notice that, in the flavor of functional programming, iterating over list elements is encapsulated in particular functions (here ZipWith and Count). The DSL does not include explicit conditional statements nor loops. Its complete specification can be found in Appendix F of [2].

2.2 Method workflow

Given the above DSL, DeepCoder expects a synthesis task to be formulated in the same way as GP, i.e., as a set of training examples T (tests), for which a program should be synthesized. Each test $t \in T$ is an (in, out) pair, where *out* is the output that the program should produce for *in*. Given T, DeepCoder proceeds in two largely independent stages.

Firstly, T is fed into a previously trained feed-forwarded neural network (to be detailed in the sections that follow). In the output layer of the network, each unit is associated with one of the instructions in the DSL L, which together are interpreted as the probability distribution $p(L|T)$ of particular DSL's instructions occurring in the target program.

In the second phase, DeepCoder employs the probability estimates $p(L|T)$ to prioritize the search policy. The choice of a search algorithm to be prioritized is largely unconstrained. In [2], the authors considered: (i) depth-first search (DFS) up to certain maximum program length, which starts from an empty program and successively adds the instructions from L with respect to decreasing $p(L|T)$; (ii) "sort and add" enumeration, which maintains a set of *active* instructions (initially empty) and performs DFS only with them, in case of failure extending the set with the next instruction with the highest $p(L|T)$ and restarting the search; (iii) program sketching [14], which uses an SMT (Satisfiability Modulo Theories) solver to choose instructions and can be biased in a similar way as "sort and add"; (iv) λ^2 algorithm [5], an approach combining enumerative search and deduction, also prioritized like "sort and add". In an experiment, each of these algorithms was compared against its respective baseline, which, rather than using the estimate of $p(L|T)$ produced by the network, relied on the a priori probability of instructions in the training set of programs $p(L)$, estimated by counting the incidence of instructions in the training set of programs (to be detailed later). All search algorithms observed manyfold speedups, ranging from 2x to 907x, depending on the assumed maximum program length and available computational budget [2][2].

2.3 Network architecture

The neural architecture proposed by Balog et al. [2] is relatively straightforward, and, its name to the contrary, is not necessarily *deep* by today's standards. The model is a feedforward layered network, with the input layer of appropriate size to accommodate for the information on input-output examples. For any particular input-output example (in, out), the information of both values and

[2]However, it is worth noting that the authors of [2] do not report the speedups on individual benchmarks, but the speedups obtained on solving a given percentile of easiest (fastest to solve) benchmarks.

types is encoded before feeding into the network, for both *in* and *out* (note for instance that in the example in Listing 1, *in* comprises a *pair* of lists). For each value, its type is encoded separately via one-hot encoding, which requires two inputs (one for the type int, one for [int]). The values themselves are always 'cast' to the list type, i.e. the scalars (int) are encoded as lists of length 1. The length of lists is limited to 20; if a list is shorter, the remaining elements are padded with a special value NULL (technically implemented as the value 256).

Each individual int value is constrained to the interval $[-256, 255]$ and not fed directly to the network, but first embedded in a 20-dimensional space (the embedding process being also part of the gradient-based backpropagation training of the network). The special value NULL is also subject to that embedding. Thus, each scalar int in (in, out) translates into 20 inputs to the network.

As a result, each value (list or scalar) requires $2+20\times20 = 402$ inputs to the network (type encoding plus embedding dimensionality). Given that the considered programs have input arity at most 2 (two values) and produce one output value, this totals to $3 \times 402 = 1206$ inputs. The input layer is followed by three consecutive nonlinear (sigmoid) layers, 256 units each. The outputs of the last of those layers, computed independently for each input-output example, are then averaged and linearly mapped (in a fully-connected manner) to the output layer comprising $|L| = 34$ units, which each of them intended to predict the likelihood of the corresponding instruction in the DSL.

2.4 Generation of the training set

To provide possibly accurate estimates of the $p(L|T)$ probabilities, the network is trained on a sample of programs, each accompanied with some input-output examples that characterize its behavior.

The training set of programs $\mathcal{T}$ is built by enumerating the programs in the DSL, starting from the shortest one-instruction programs and increasing their length up to $l_{\max}$. To avoid redundancy, programs that create variables which do not affect program output are discarded. Also, each newly created program is checked for semantic equivalence with the ones already collected in $\mathcal{T}$ by inspecting the outputs it produces for a set of examples. Should such an equivalence hold, the new program is discarded. In this way, shorter programs with a given input-output behavior are preferred.

For each program $P \in \mathcal{T}$, a corresponding set of (in, out) examples T is then created. This is in general nontrivial, as randomly chosen inputs are unlikely to evoke useful diversification of outputs, or may cause programs to terminate due to errors (for instance, the program in Listing 1 fails for any pair of input lists $in = (a, b)$ that have different lengths, due to the ZipWith operation). Therefore, program *output* is randomly drawn first, and then 'back-propagated' through the program, resulting with the constraints on input variables. From those constraints, inputs' values are then uniformly drawn.

As a result of the above, one obtains a training set of pairs

$$((in_1, P(in_1)), \dots, (in_5, P(in_5)); P)$$

where $P \in \mathcal{T}$ is a program, and $(in_i, P(in_i))$ are (in, out) examples illustrating P's behavior on input data. For the DSL considered here and $l_{\max}$ set to 3, $|\mathcal{T}| = 822{,}582$; for $l_{\max} = 4$, $|\mathcal{T}| = 5{,}004{,}532$.

2.5 Network training

The neural network model described in Section 2.3 is trained on the generated training set. For each training example, the input-output examples $(in_i, P(in_i))$ are fed into network inputs using the encoding described earlier. P is encoded as a binary vector of length $|L| = 34$, with 1s indicating instructions that occur in P and 0s those absent, and such a vector forms the corresponding desired output of the network.

As the neural network architecture and training protocol used in this paper diverge from the ones in the original DeepCoder, we provide the remaining details in the experimental section.

3 THE PROPOSED APPROACH

We propose to use the predictions made by DeepCoder-like neural networks to bias program synthesis performed by GP. Similarly to DeepCoder, we assume that a network mapping examples T to a probability distribution of instructions in L has been trained on some training set $\mathcal{T}$ (Section 2.4).

Given a specific synthesis task T in the form of (in, out) examples, we first query the network on it, obtaining the estimated probability distribution $p(L|T)$. That distribution is subsequently used to bias the search conducted by GP, for which we consider the following two variants of such 'priming':

Search-only (S): In this variant, we parameterize the mutation operator with a probability distribution p to be used when replacing the instructions in programs with new, randomly chosen instructions. In the default, unbiased variant (typical for standard GP), p is uniform over all instructions. In the proposed approach, $p = p(L|T)$, causing the mutation operator to use the instructions that are found plausible by the network more often than others.

Init-and-search (IS): In this variant, we parameterize both the mutation operator (as in the Search-only variant) *and* the population initialization. In analogy to the priming of the mutation operator described above, the algorithms used to produce the initial population of programs use $p(L|T)$ rather than the uniform distribution over all instructions.

The details are provided in the following description of search operators, which are tailored to the specifics of the considered DSL and, among others, have to maintain type- and reference consistency.

Program representation. We represent programs as fixed-length sequences of instructions, each issuing a function call and creating a fresh variable that stores the result of that call. The length of all programs in a given GP run is the same; however, the *effective length* of the program can be smaller due to the availability of the Nop operation.

As signalled earlier, the number of the input arguments and their types are assumed to be given as part of problem specification, and thus the initial instructions that fetch those arguments (e.g., the first two instructions in Listings 1 and 2) are not explicitly represented in the programs. As a consequence, they cannot be also affected by the search operators described in the following.

Program initialization. Programs are initialized instruction by instruction. For each instruction, a function f is selected from

L according to the assumed probability distribution p (e.g, $p = p(L|T)$). Next, the algorithm examines the signature of f, and verifies whether arguments of the appropriate types are available at this stage of program execution (i.e., have been created by the previous instructions). Should that be the case, the variables are drawn from the scope and passed to f. If f requires other arguments (i.e., predicates and lambdas listed in Section 2.1), those are also drawn according to p. If variables of required type are not available, a new function is drawn for this instruction and the process is repeated[3]. This is continued until the assumed program length is reached.

Mutation. Mutation picks an instruction in a program at random, analyzes the function call $f()$ issued there, and determines the subset of functions in the DSL that have the same function signature as f. Subsequently, we normalize a probability distribution p of these functions by dividing each value by their sum. Then, a function is drawn from that subset according to p, and it replaces the original one.

We considered also two alternative designs of mutation operators, which however proved inferior in preliminary experiments, so we do not present them here.

Crossover. A typical crossover operator creates new individuals by combining parts from two parents so that the offspring inherits some of their traits. Following this design, we implement a fairly straightforward crossover operator that creates the offspring by exchanging up to l_c compatible function calls in parent programs (in our experiments $l_c = 2$). On a technical note, we first identify all possible crossover points by extracting type signatures for successive l_c-sized blocks (sublists) of instructions and narrowing the list down to the signatures that co-occur in both parents. Next, we randomly pick a signature that allows performing type-safe exchange and guarantees syntactic correctness of the resulting offspring. If more than one block of instructions complies to the chosen signature, the choice is made randomly. Finally, if there are no such blocks at all, we repeat the procedure with $l_c - 1$, or return the parent programs when l_c has reached zero.

Note that the crossover operator is not affected by p in any way.

4 RELATED WORK

By combining evolutionary synthesis with neural networks, this study brings together two paradigms and relates to a range of studies in both fields.

In the field of **neural networks**, the recent dawn of deep learning manifested also in the area of program synthesis. As a result, a range of approaches have been proposed that make use of the neural paradigm for the purpose of constructing various executable structures. Notable achievements include neural interpreters that mimic Turing machines [6] or differentiable interpreters, like that for the programming language Forth [3]. In a similar spirit, Zaremba et al. showed in [15] how neural reinforcement learning can learn simple algorithms that manipulate data on one-dimensional tapes and two-dimensional grids.

The contributions cited above explore the possibilities of *reproducing* the functionalities typical for interpreters of programming languages in neural substrates. Though undoubtedly interesting, these are largely proof-of-concepts with limited practical implications, because the goal of program synthesis is to produce programs in human-readable programming languages, which preferably have been already accepted by the community of developers. This need is addressed by DeepCoder [2], the reference approach for this study, which we presented in detail in Section 2: a neural network is used to guide a synthesis algorithm that yields programs that are readable for humans, can be inspected, reasoned about, and the formal correctness of which can be potentially proven. As we argue later in Section 6, using similar neural guidance for other programming languages than the DSL considered here (e.g., subsets of popular programming languages) is entirely plausible.

In [4], Devlin et al. propose to call the approaches in the former group *neural program induction*, while those in the latter *neural program synthesis*. They also briefly review more works that fall into both these categories. Last but not least, they confront two approaches, each representing one of these categories, on a demanding real-world task of synthesizing string manipulation programs meant to capture string transformation patterns provided in examples, as posed in the seminal work by Gulwani et al. [7] and currently available as the FlashFill feature in Microsoft Excel^{TM}. Interestingly, the authors demonstrate that the neural approaches are much more resistant to the presence of noise in the training data (e.g., typos) than the original FlashFill, the feature which is of particular importance in this application area.

With respect to **evolutionary program synthesis**, the approach proposed here can be seen as genetic programming supported by a machine learning model. Several studies in the past employed machine learning support to enhance GP, and most of them concerned fitness function. The arguably most natural approach is to use machine learning models (regression models) as a *surrogate fitness* (see, e.g., [10, 13]), in order to circumvent the potentially costly exact fitness evaluation. However, machine learning can be helpful not only to simulate the original fitness function, but also to provide some alternative/additional guidances. For instance, [12] proposed to use a decision tree to both augment the fitness function (by introducing additional search objectives) and select potentially useful pieces of code from the evaluated individuals, in order to reuse them in search operators.

5 EXPERIMENT

We devise a controlled experiment aimed at assessing the impact of the guidance provided by a trained neural network on GP's search performance, and compare the two variants described in Section 3, i.e. Search-only (**S**) and Init-and-search (**IS**), with the following two baseline configurations:

(1) **U:** An unbiased GP search, i.e. with each instruction having the same probability of being used by the mutation operator (uniform p).
(2) **P:** A GP search with the mutation operator biased with the *a priori* probabilities of instructions, i.e. their frequency of occurrence in the training set $\mathcal{T}$ ($p = p(L)$).

Notice that both these configurations use the provided distribution only in mutation; the probability of instructions in population initialization is in both cases uniform.

[3]The DSL has the property that for any set of variables, at least one function exists that is applicable to (some of) them, so this loop is guaranteed to eventually terminate.

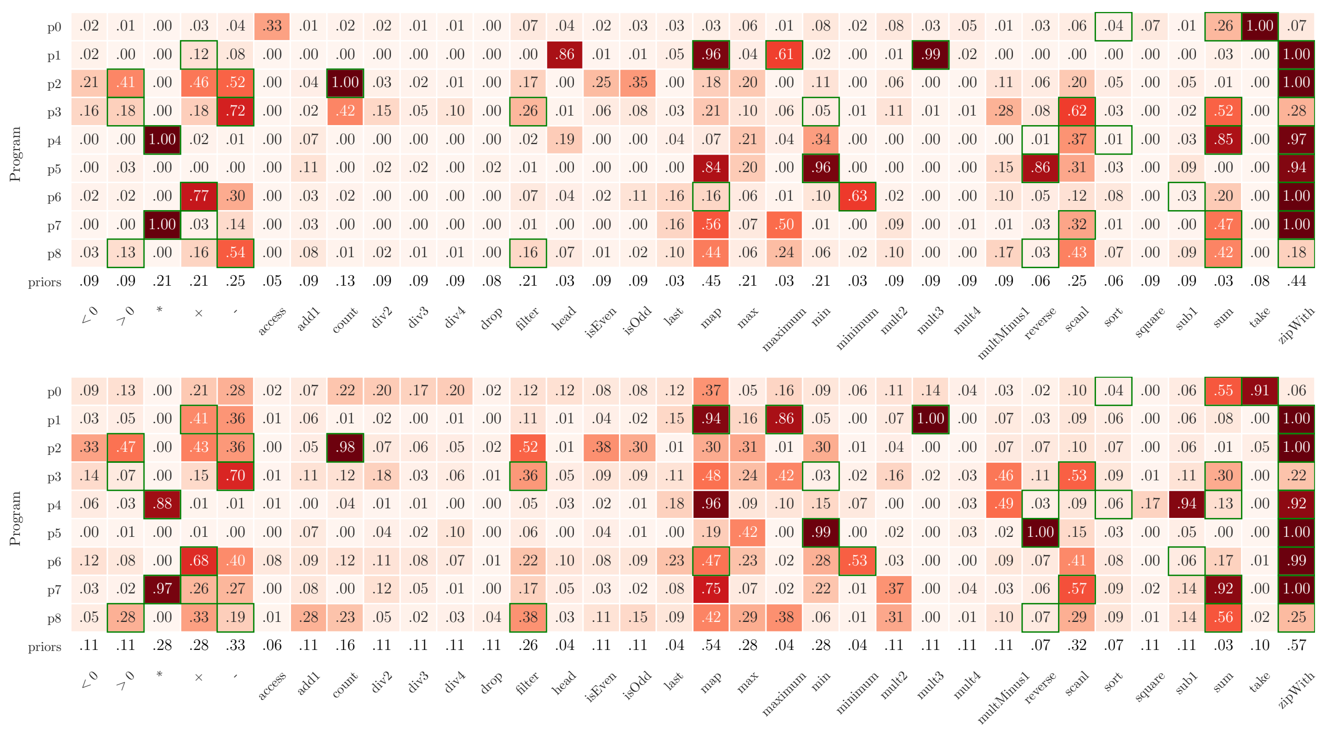

Figure 1: Visualization of networks' outputs for individual benchmarks.

All compared methods implement the generational evolutionary algorithm without elitism. Given the fixed-length, linear program representation, we adopt the operator pipeline typical for genetic algorithms: a selection operator is first applied twice to the current population, to fetch a pair of parent programs. The crossover operator is then applied individually to the parent programs at probability p_c; otherwise, the parents are left untouched. Then, each of the resulting programs is subject to mutation with probability p_m. Finally, the resulting two programs are added to the next population.

As signalled earlier, we run two series of experiments: for the **small training set** of 822,582 programs of maximum length $l_{\max} = 3$, and the **large training set** of 5,004,532 programs for $l_{\max} = 4$.

5.1 Benchmarks

To assess the generalization ability of the proposed approach, we use the benchmarks considered in [2]. We describe their semantics only briefly here, and refer the interested reader to Appendix A in the cited work for their source code. P0(int,[int]) sums the requested number of the smallest elements from the list. P1([int],[int]) computes the maximum score of a team in a soccer league based on the number of wins and ties. P2([int],[int]) counts the number of list elements that are greater in the first argument than in the second argument (Listing 1). P3([int]) calculates the total difference of list elements with respect to the smallest element. P4([int],[int]) returns the minimal area of rectangles of dimensions given in the input lists (Listing 2). P5([int]) calculates the list of the minima of the input list and the reversed input list. P6([int],[int]) calculates the minimum of the list of the sum of the input lists, decreased by 2. P7([int],[int]) calculates a cumulative expression over two input lists zipped together, and P8([int]) similarly for a single input list, involving also certain filtering of the intermediate results.

To sum up, the arity of the benchmark programs varies from 1 to 2, and they implement three different signatures: ([int]), (int,[int]), and ([int],[int]). The output type is int or [int]. The effective length ranges from 2 to 5; more precisely:

Benchmark	P0	P1	P2	P3	P4	P5	P6	P7	P8
Length	3	3	2	4	5	2	4	3	4

Note therefore that only P0, P1, P2, P5, P7 are guaranteed to be present in both small and large training set; notably, P4 is absent also from the large training set. Though we could have removed the benchmarks from the training set in order to adhere to the strict training-testing set division, we anticipate that dropping just a handful of programs out of almost a million (small training set) or over 5 million (large training set) of them (all semantically unique) would almost certainly have no measurable effect. A more detailed analysis of the generalization power, possibly using large samples of random benchmarks (for instance taken out from the current training sets) is left for future work.

5.2 Training of the neural network

We initialize networks' weights with the He method [8], as implemented in the TensorFlow software library [1]. All considered training algorithms are variants of stochastic gradient descent and error backpropagation, and operate very similarly: examples are forward-propagated, in batches of 512 examples, through the network structure, then errors calculated as the cross-entropy between the actual and desired output(s), propagated backwards, and based on that units' weights are updated according to delta rule. We used two specific algorithms subscribing to this scheme: RMSProp and Adam [11]. Training lasts up to 100 epochs (full passes over the training set $\mathcal{T}$) with early stopping condition occurring when validation loss ceased to improve.

We also considered slight variations wrt the original DeepCoder architecture described in Section 2.3. Firstly, given that the occurrences of instructions in a program are largely unrelated, we abandon the original assumption that the last layer of the network should represent log-unnormalized probabilities (logits), and thus use sigmoid activations in that layer and binary cross-entropy as the loss function. We also replace the original sigmoidal units in the hidden layers with rectified linear units (ReLUs), 'leaky' ReLUs, and exponential-linear units (ELUs). Approximately a dozen of such architectures have been tested for both the small training set and the large one. In both groups, the networks trained with the Adam algorithm typically ranked at the top with respect to accuracy on the test set (10,000 programs not present in the training set), in particular when combined with ReLU or ELU units. Ultimately, we selected the ReLU-based architecture for the small dataset (accuracy 92.48%), and the ELU-based one for the large training set (accuracy 90.85%), both trained with Adam (those networks committed also the smallest error measured with the Hamming distance from the vector of desired outputs: 0.0752 and 0.0915, respectively).

High accuracy and low Hamming distance suggest that the networks manage to make largely accurate predictions. This is indeed confirmed in Fig. 1, which visualizes the outputs produced by the two above-mentioned best-performing networks for individual benchmarks (in rows): the top heatmap for the network chosen for the small training set, the bottom heatmap for the one selected for the large training set. The columns correspond to the $|L| = 34$ instructions (or more generally, lexical elements) of the DSL. The numbers and shading report the probability estimates produced by the network. The green frames mark the instructions that actually occur in the target programs. At the bottom of the both heatmaps, we provide also the priors, i.e., the probabilities of instructions' occurrences in the entire training set.

The networks seem very responsive to the characteristics of the examples in T: their predictions clearly diverge from the priors and vary per benchmark, sometimes positively, sometimes negatively. On the other hand, the predictions correlate with the priors on average. Comparing the heatmaps with the source codes of target programs in Listings 1 and 2, and those in Appendix A of [2], reveals that networks' high estimates often coincide with the correct instructions; in particular, such instructions are typically among those scored highest by the networks. It may be also worth mentioning that the networks clearly do not base their estimates only on benchmark's signature, i.e. the *types* of the arguments and outputs: of the nine benchmark programs considered here, only one takes a scalar (int) argument (P0), and only two produce lists (P0 and P5) – all the remaining ones expect lists as arguments and produce an int as the outcome. The networks must have thus detected some nontrivial patterns in the provided input-output examples.

Table 1: Common parameter settings for all methods.

Parameter	Value
Probability of mutation p_m	0.8
Probability of crossover p_c	0.0 or 0.5
Population size	1000
Selection method	Tournament (**T**) or Lexicase (**L**)
Max program length	3 or 4
Number of fitness cases	128
Max generations	200

5.3 GP settings and parameter tuning

We conducted preliminary parameter tuning to optimize the performance of baseline configurations, by sweeping the parameters in the following ranges: the probabilities of mutation and crossover $p_m, p_c \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, population size $\in \{100, 500, 1000\}$, and running each configuration 30 times on the representative sample of four benchmarks: P0, P1, P3, P4. Analysis of resulting success rates and fitness graphs led us to choosing the settings shown in Table 1 as optimal and using them in the remaining part of this paper for all methods. In particular, we compare setups with ($p_c = 0.5$, marked with **C**) and without ($p_c = 0$, marked with **N**) crossover operator, to verify whether it contributes to the actions of the mutation biased by p.

The length of the programs in the population is set to 1 plus the length of the target program. Alongside with Tournament selection (with tournament of size 7, **T**), we consider also Lexicase selection (**L**) [9], which does not rely on scalar fitness for selection, and instead takes into account the individual outcomes of *interactions* between programs and tests (examples). In each selection act, a random permutation of tests is generated, and the program from the current population which passes the longest uninterrupted sequence of tests is selected. In multiple studies, Lexicase proved systematically better than Tournament selection.

Though the neural network predicts the $p(L|T)$ based only on the five examples in T, this set is insufficient to define a fine-grained fitness function. For that purpose, we draw an additional set of 128 tests T', using the same procedure as for the tests in T (Section 2.4). The fitness function is minimized and counts the number of errors committed by programs on the tests in T'.

A GP run is terminated if it finds the correct program before the 200 generations elapse. However, program correctness is judged based on the fitness alone, i.e. with respect to the tests in T' – there is no guarantee that the program is correct outside of the provided sample input-output examples.

5.4 Results

The primary performance indicator we consider is the success rate, i.e., the percentage of runs that ended up with a correct program, out of 50 runs. We report it in Tables 2 and 3, respectively for the networks trained on the small and large training set, with and without crossover, and Tournament selection vs. Lexicase selection. We also rank the methods in each group, on each benchmark separately, average the ranks, and report them in the last row of the tables (the resulting ranks must range thus in $[1, 4]$). Notice that

Table 2: Success rates for particular configurations, for the small training set. Legend: T (tournament), L (lexicase), U (unbiased), P (priors baseline), S (search), IS (initialization and search).

method	T_{U}		T_{P}		T_{S}		T_{IS}		L_{U}		L_{P}		L_{S}		L_{IS}		mean
cx	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	
P0	0.70	0.54	0.34	0.42	0.88	0.94	**1.00**	**1.00**	0.58	0.66	0.40	0.58	0.72	0.82	**1.00**	**1.00**	0.72
P1	0.18	0.16	0.26	0.24	0.24	0.20	0.54	0.58	0.16	0.08	0.20	0.12	0.60	0.44	**0.96**	**0.96**	0.37
P2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.00
P3	0.14	0.16	0.12	0.12	0.46	0.48	**1.00**	0.96	0.52	0.62	0.28	0.54	0.82	0.76	**1.00**	**1.00**	0.56
P4	0.14	0.06	0.02	0.08	0.02	0.02	0.00	0.00	0.52	**0.56**	0.38	0.44	0.38	0.18	0.14	0.14	0.19
P5	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.98	0.92	**1.00**	0.98	**1.00**	0.98	**1.00**	**1.00**	0.99
P6	0.08	0.08	0.06	0.14	0.02	0.14	0.04	0.04	0.40	0.60	**0.82**	0.68	0.68	0.74	0.78	0.80	0.38
P7	0.16	0.08	0.34	0.16	0.34	0.44	0.56	0.58	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.67
P8	0.18	0.36	0.10	0.12	0.14	0.18	0.28	0.32	0.36	0.46	0.26	0.30	0.50	0.34	**0.82**	0.76	0.34
mean	0.40	0.38	0.36	0.36	0.46	0.49	0.60	0.61	0.61	0.66	0.59	0.63	0.74	0.70	0.86	0.85	

Table 3: Success rates for particular configurations, for the large training set.

method	T_{U}		T_{P}		T_{S}		T_{IS}		L_{U}		L_{P}		L_{S}		L_{IS}		mean
cx	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	
P0	0.70	0.54	0.34	0.38	0.82	0.78	**1.00**	**1.00**	0.58	0.66	0.54	0.58	0.64	0.68	**1.00**	**1.00**	0.70
P1	0.18	0.16	0.20	0.20	0.18	0.24	0.58	0.62	0.16	0.08	0.16	0.16	0.48	0.32	**0.98**	0.88	0.35
P2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.00
P3	0.14	0.16	0.10	0.10	0.12	0.28	0.68	0.74	0.52	0.62	0.46	0.52	0.60	0.74	**0.98**	0.94	0.48
P4	0.14	0.06	0.02	0.00	0.02	0.02	0.00	0.00	0.52	**0.56**	0.52	0.50	0.22	0.32	0.32	0.08	0.21
P5	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.98	0.92	0.98	0.96	**1.00**	**1.00**	**1.00**	**1.00**	0.99
P6	0.08	0.08	0.00	0.04	0.08	0.10	0.12	0.12	0.40	0.60	0.64	0.70	0.64	0.70	0.72	**0.74**	0.36
P7	0.16	0.08	0.28	0.24	0.48	0.42	0.78	0.90	**1.00**	**1.00**	0.98	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.71
P8	0.18	0.36	0.16	0.08	0.18	0.24	0.42	0.42	0.36	0.46	0.32	0.34	0.28	0.32	**0.84**	0.78	0.36
mean	0.40	0.38	0.34	0.34	0.43	0.45	0.62	0.64	0.61	0.66	0.62	0.64	0.65	0.68	0.87	0.82	

the configurations that use uniform distributions do not rely on the training set and thus produce the same values in both tables.

The benchmarks vary in the level of difficulty they pose to the methods, ranging from very easy ones, which are solved in all runs by all methods (P2), to difficult ones, on which even the best performing configurations barely exceed 50% probability of success (P4). Unsurprisingly, success rate seems to negatively correlate with the length of the target program (2 for P2 (Listing 1) and 5 for P4 (Listing 2), the longest target program in the benchmark suite).

Most importantly, the success rates of the configurations parameterized with networks' estimates $p(L|T)$, i.e. S and IS, are systematically better than those of configurations that rely on the uniform distribution (U) and those parameterized by the prior probabilities calculated from the training test (P). Interestingly, the latter is usually worse than the former, which suggests that for an approach that is not informed by the network, it is better to use the uniform distribution. The possible explanation is that some instructions have very low occurrence in the training sets, and it thus becomes very unlikely for them to be used, even when they are required for a given task.

Of the two approaches informed by networks, priming both initialization and search (IS) performs better. Though this effect was expected, we did not anticipate its size (note the large differences between the average ranks of IS and S configurations). We hypothesize that the observed gain stems from the fact that initialization is relatively likely to produce a program that uses all instructions appointed as most probable by the network. In contrast, the mutation operator can substitute only one instruction at a time, so if it happens to start with a program that contains no useful instructions, it needs to be applied several times to produce the desired effect, which is unlikely.

The Lexicase selection operator (L setups) proves its usefulness again, systematically and significantly boosting the success rates in comparison to the Tournament selection (T setups). Nevertheless, in relative terms, the informed configurations improve over the non-informed ones irrespectively of the type of selection operator, which suggests that priming is beneficial independently of this component of search algorithm. It becomes thus even less likely for the observed effects of priming to be incidental.

What comes as a bit of surprise is the not so clearly positive effect of using the large training set, when compared to the small one. For IS, moving from the latter to the former causes the success rate to improve only in 11 cases (combinations of benchmark and settings), out of the total of $9 * 4 = 36$. At the same time, we observe deterioration in 8 cases, and in the remaining 17 cases the success rate does not change. Similarly, there is no clear winner when comparing the small and the large dataset for S configurations.

To statistically evaluate our results, we employed the Friedman's test for multiple achievements of multiple subjects. In Table 4 we present the average ranks and p-values computed for four disjoint groups of configurations:

Table 4: Ranks for the tested configurations. Legend: small-/large (training set used), N (no crossover), C (crossover).

small$_N$ ($p = 0.00877$)

Method	L$_{IS}$	L$_S$	T$_{IS}$	L$_P$	L$_U$	T$_S$	T$_U$	T$_P$
Rank	2.50	3.06	4.28	4.28	4.56	5.50	5.67	6.17

small$_C$ ($p = 0.010579$)

Method	L$_{IS}$	L$_S$	T$_{IS}$	L$_U$	L$_P$	T$_S$	T$_U$	T$_P$
Rank	2.17	3.61	4.33	4.33	4.72	5.22	5.72	5.89

large$_N$ ($p = 0.00093$)

Method	L$_{IS}$	L$_S$	T$_{IS}$	L$_U$	L$_P$	T$_S$	T$_U$	T$_P$
Rank	2.06	3.61	3.72	4.44	4.83	5.44	5.50	6.39

large$_C$ ($p = 0.00075$)

Method	L$_{IS}$	L$_S$	T$_{IS}$	L$_U$	L$_P$	T$_S$	T$_U$	T$_P$
Rank	2.22	3.50	3.83	4.33	4.56	5.11	5.83	6.61

- **small**$_N$ – small training set, methods not using crossover.
- **small**$_C$ – small training set, methods using crossover.
- **large**$_N$ – large training set, methods not using crossover.
- **large**$_C$ – large training set, methods using crossover.

In each group, the configurations that rely on neural guidance clearly rank at the top. The p-values indicate that some methods are significantly better than others. We conducted a post-hoc analysis, which let us conclude that L$_{IS}$ is statistically significantly better than T$_U$ and T$_P$ in every group. Additionally, in the large$_N$ group, L$_{IS}$ was significantly better than T$_S$.

6 DISCUSSION

It is worth emphasizing that the proposed approach is largely independent from the underlying programming language (DSL). In this study, we used the DSL from [2] mainly to provide for some degree of comparison with the results presented there. There are no obstacles for using linear programs written in other DSLs, or tree-based GP; for the network, that would require only adjusting the size of the output layer to the number of instructions in the DSL.

Unavailability of the original implementation of DeepCoder [2] prevents us from conducting a side-by-side comparison with it. We posit however that using neural priming for *stochastic* search can be particularly beneficial. Our argument is that neural estimates of probabilities are inherently noisy (not least because they are based on just a handful of examples), and thus treating them with absolute confidence bears certain risks – and this is what, at least in a certain sense, some of the deterministic search algorithms considered in [2] do. In particular, the depth-first search uses the instructions strictly in the ordering given by the estimated probabilities, so an instruction unfairly deemed as unlikely will wait very long to be used. We suppose that this deficiency was one of the reasons for which the authors of DeepCoder devised the "sort and add" heuristics (Section 2). A stochastic search, like the evolutionary algorithm considered here, is free from that shortcoming, as it treats the estimates as probabilistic guidance only.

7 CONCLUSION

In this paper, we evidenced the possibility of augmenting the guidance of evolutionary program synthesis with a neural network that estimates the likelihoods of instructions based on the samples of input-output behavior. The resulting boosts in success rate seem promising, and it is certainly possible to make them statistically more significant by extending the benchmark suite and/or further extensions of the method. Concerning the latter, possibilities galore. For instance, the current implementation of mutation takes into account only the desired (estimated) distribution of instructions, while entirely ignoring the instructions already present in the program; one could easily devise a search operator that would prefer inserting the instructions that are indicated as likely by the network *and* still absent from the program. Next, we use the estimates to prime the search operators only; one could consider priming also the fitness function (or adding an extra objective) by inspecting the source code of candidate programs and promoting those that feature the desired instruction. On the more analytical side, it would be interesting to see how the method generalizes with program length, and how its performance degrades when considering more complex benchmarks.

Acknowledgment The computations were performed in Poznan Supercomputing and Networking Center. The authors acknowledge support from Polish National Science Centre through the following grants: PL – 2014/15/N/ST6/04572; IB, KK – 2014/15/B/ST6/05205.

REFERENCES

[1] Martín Abadi and et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *arXiv preprint arXiv:1611.01989* (November 2016). https://arxiv.org/abs/1611.01989

[3] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel. 2016. Programming with a Differentiable Forth Interpreter. *ArXiv e-prints* (May 2016). arXiv:1605.06640

[4] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. *ArXiv e-prints* (March 2017). arXiv:cs.AI/1703.07469

[5] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[6] A. Graves, G. Wayne, and I. Danihelka. 2014. Neural Turing Machines. *ArXiv e-prints* (Oct. 2014). arXiv:1410.5401

[7] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/doi:10.1145/2240236.2240260

[8] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[9] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. https://doi.org/doi:10.1109/TEVC.2014.2362729

[10] Torsten Hildebrandt and Juergen Branke. 2015. On Using Surrogates with Genetic Programming. *Evolutionary Computation* 23, 3 (Fall 2015), 343–367.

[11] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[12] Krzysztof Krawiec and Una-May O'Reilly. 2014. Behavioral programming: a broader and more detailed take on semantic GP. In *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*, Christian Igel, et al. (Ed.). ACM, Vancouver, BC, Canada, 935–942. https://doi.org/doi:10.1145/2576768.2598288 Best paper.

[13] Pawel Liskowski and Krzysztof Krawiec. 2016. Surrogate Fitness via Factorization of Interaction Matrix. In *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming (LNCS)*, Malcolm I. Heywood, James McDermott, Mauro Castelli, Ernesto Costa, and Kevin Sim (Eds.), Vol. 9594. Springer Verlag, Porto, Portugal, 68–82. https://doi.org/doi:10.1007/978-3-319-30668-1_5

[14] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Electrical Engineering and Computer Science, University of California, Berkeley, USA. http://people.csail.mit.edu/asolar/papers/thesis.pdf

[15] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. 2015. Learning Simple Algorithms from Examples. *ArXiv e-prints* (Nov. 2015). arXiv:cs.AI/1511.07275