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

Paweł Liskowski, Iwo Błądek, Krzysztof Krawiec

Institute of Computing Science, Poznań University of Technology

9 October 2018

Outline

Intro

In this presentation, we will describe work originally presented at GECCO 2018 conference in Kyoto, 15-19.07.2018.

This work is also described in the peer-reviewed publication:

[2] P.Liskowski, I.Błądek, K.Krawiec, *Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks*, GECCO'18 Proceedings of the Genetic and Evolutionary Computation Conference, ACM, 2018, pp. 1143-1150.

Program synthesis

Automatic program synthesis: a general class of problems where the goal is to find a program (executable procedure) that satisfies a given specification.

Specification

```
2, [1 5 3]  →  4
1, [1 8 3 5]  →  1
1, [1 8 3 5 7]  →  1
2, [1 8 3 5 7]  →  4
3, [1 8 3 5 7]  →  9
2, [1 5 3 0 8]  →  1
...
```

Target program

```
a ← int
b ← [int]
c ← Sort b
d ← Take a c
e ← Sum d
```

Aspects: type of specification, programming language.

Genetic programming

- Population of candidate programs.
- In each generation programs are being selected based on their *fitness* and *search operators* modify those programs, which then constitute a new population.

Example of search operators:

Mutation:

```
(ite (>= x y) 2 (+ x y))
              ↓
(ite (< y 0) 2 (+ x y))
```

Crossover:

```
(mod x 2)
(ite (>= x y) 2 (+ x y))
              ↓
(mod (+ x y) 2)
(ite (>= x y) 2 x)
```

Motivation

Problem we wanted to solve:

- Search operators work under assumption that every instruction has the same chance to lead to a correct candidate program (**uniform distribution of instructions given the problem instance**).
- In practice, this in vast majority of cases **does not hold**.

Our contribution:

- Search operators (mutation, population initialization) taking into account the **conditional probability of instructions** given input-output examples from the specification.
- Conditional probability of instructions is obtained by **training a neural network** on input-output examples.

Outline of our approach

“All” problem instances

1. Train artificial neural network (NN) to estimate conditional probability of program instructions given the I/O examples.

Outline of our approach

"All" problem instances

1. Train artificial neural network (NN) to estimate conditional probability of program instructions given the I/O examples.

Particular problem instance

1. Query the neural network on the I/O examples to obtain probability estimates.
2. Parametrize search operators (mutation, population initialization) of GP with the obtained estimates.
3. Run GP.

Discussion

Artificial neural network is used, but should the whole proposed solution be treated as a classical machine learning scenario?

Tentative answer: No. Machine learning subcomponent is used to guide search, but in the end this is a search problem.

Outline

1. Introduction
2. **Search problem**
3. Training neural network
4. Neuro-Guided Genetic Programming
5. Experiments

Search problem

- **Goal:** find such a program in the programming language (*DeepCoder DSL*) that the specification will be met.

- **Specification:** a list of input-output examples.

DeepCoder DSL

Types:

- `Int`
- `List[Int]`

Functions:

- (10) operations on lists: HEAD, LAST, TAKE, DROP, ACCESS, MINIMUM, MAXIMUM, REVERSE, SORT, SUM
- (5) higher-order functions: MAP, FILTER, COUNT, ZIPWITH, SCANL1

Other elements of the language:

- (10) lambdas for MAP (ADD1, SUB1, MULTMINUS1, MULT2, MULT3, MULT4, DIV2, DIV3, DIV4, SQUARE).
- (4) predicates for FILTER and COUNT (>0, <0, ISODD, ISEVEN).
- (5) lambdas for ZIPWITH and SCANL ($+$, $-$, $*$, MIN, MAX).

DeepCoder DSL

We use the same DSL as was used in the DeepCoder paper [1].

Program representation:

- A variant of linear GP.
- A fixed-length sequence of instructions, each of which issues a function call, and stores it's result in a fresh variable.

Example program:

P0: Compute the sum of *a* smallest numbers from the list *b*.

```
a ← int
b ← [int]
c ← Sort b
d ← Take a c
e ← Sum d
```

Input:
`2, [1 8 3 5 7]`
Output:
`4`

Program representation – inputs

```
a ← int
b ← [int]
```

Declaring program's input. Variable *a* will be an arbitrary *Int* provided by the user, and *b* will be an arbitrary *List[Int]*.

Program representation – functions

```
a ← int
b ← [int]
c ← Function {a, b}+
d ← Function {a, b, c}+
e ← ...
```

Every line of the program consists of a single application of a function to the previously defined variables.

For example:

```
a ← int
b ← [int]
c ← Sort b
d ← Take a c
e ← Sum d
```

Program representation – predicates

```
a ← int
b ← [int]
c ← Function predicate {a, b}⁺
d ← Function lamba {a, b, c}⁺
e ← ...
```

Some functions accept certain predicates or lambdas, which are predefined and treated as constant elements of the language.

For example (lambdas in red):

```
a ← [int]
b ← [int]
c ← Map (*3) a
d ← ZipWith (+) c b
e ← Maximum d
```

Example problems

P0: Compute the sum of *a* smallest numbers from the list *b*.

```
a ← int
b ← [int]
c ← Sort b
d ← Take a c
e ← Sum d
```

Input:
`2, [1 8 3 5 7]`
...
Output:
4

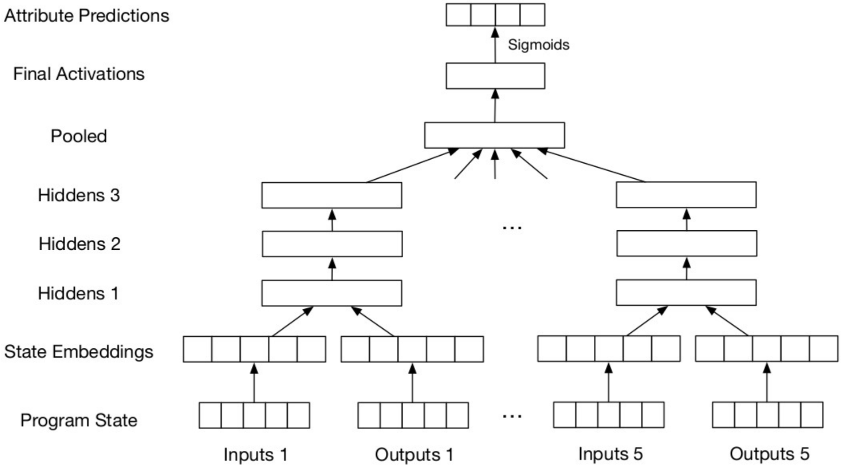
P4: Given lists *a* and *b*, compute the minimal area of rectangles of dimensions given in the input lists.

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

Input:
`[1 2 3], [4 5 6]`
...
Output:
28

Outline

Network architecture



Source: M.Balog, et al., *"DeepCoder: Learning to Write Programs"*, 2016, https://arxiv.org/abs/1611.01989

Network training

- Training algorithm: Adam.
- Training lasts up to 100 epochs (full passes over the training set).
- Early stopping condition: validation loss ceased to improve.

Generation of the training set

- All programs up to a certain number of instructions while removing most semantic duplicates.
- Each training case is a tuple (I/O-examples, instructions vector).
- I/O-examples are generated randomly.

- ***Small training set*** – programs up to length 3 with most of the semantic duplicates removed. Total count: 822,582 programs.

- ***Large training set*** – programs up to length 4 with most of the semantic duplicates removed. Total count: 5,004,532 programs.

Test sets

- 10,000 programs not present in the training set.
- Several neural architectures and learning algorithms were tested and we selected the one with the highest accuracy on the test set.

training set	total programs	accuracy on test set (%)
small	822,582	92.48
large	5,004,532	90.85

Heatmaps

Small training set:

Program	<0	>0	*	×	-	access	add1	count	div2	div3	div4	drop	filter	head	isEven	isOdd	last	map	max	maximum	min	minimum	mult2	mult3	mult4	multMinus1	reverse	scanl	sort	square	sub1	sum	take	zipWith
p0	.02	.01	.00	.03	.04	.33	.01	.02	.02	.01	.01	.00	.07	.04	.02	.03	.03	.03	.06	.01	.08	.02	.08	.03	.05	.01	.03	.06	.04	.07	.01	.26	1.00	.07
p1	.02	.00	.00	.12	.08	.00	.00	.00	.00	.00	.00	.00	.00	.86	.01	.01	.05	.96	.04	.61	.02	.00	.01	.99	.02	.00	.00	.00	.00	.00	.00	.03	.00	1.00
p2	.21	.41	.00	.46	.52	.00	.04	1.00	.03	.02	.01	.00	.17	.00	.25	.35	.00	.18	.20	.00	.11	.00	.06	.00	.00	.11	.06	.20	.05	.00	.05	.01	.00	1.00
p3	.16	.18	.00	.18	.72	.00	.02	.42	.15	.05	.10	.00	.26	.01	.06	.08	.03	.21	.10	.06	.05	.01	.11	.01	.01	.28	.08	.62	.03	.00	.02	.52	.00	.28
p4	.00	.00	1.00	.02	.01	.00	.07	.00	.00	.00	.00	.00	.02	.19	.00	.00	.04	.07	.21	.04	.34	.00	.00	.00	.00	.00	.01	.37	.01	.00	.03	.85	.00	.97
p5	.00	.03	.00	.00	.00	.00	.11	.00	.02	.02	.00	.02	.01	.00	.00	.00	.00	.84	.20	.00	.96	.00	.00	.00	.00	.15	.86	.31	.03	.00	.09	.00	.00	.94
p6	.02	.02	.00	.77	.30	.00	.03	.02	.00	.00	.00	.00	.07	.04	.02	.11	.16	.16	.06	.01	.10	.63	.02	.00	.00	.10	.05	.12	.08	.00	.03	.20	.00	1.00
p7	.00	.00	1.00	.03	.14	.00	.03	.00	.00	.00	.00	.00	.01	.00	.00	.00	.16	.56	.07	.50	.01	.00	.09	.00	.01	.01	.03	.32	.01	.00	.00	.47	.00	1.00
p8	.03	.13	.00	.16	.54	.00	.08	.01	.02	.01	.01	.00	.16	.07	.01	.02	.10	.44	.06	.24	.06	.02	.10	.00	.00	.17	.03	.43	.07	.00	.09	.42	.00	.18
priors	.09	.09	.21	.21	.25	.05	.09	.13	.09	.09	.09	.08	.21	.03	.09	.09	.03	.45	.21	.03	.21	.03	.09	.09	.09	.09	.06	.25	.06	.09	.09	.03	.08	.44

Large training set:

Program	<0	>0	*	×	-	access	add1	count	div2	div3	div4	drop	filter	head	isEven	isOdd	last	map	max	maximum	min	minimum	mult2	mult3	mult4	multMinus1	reverse	scanl	sort	square	sub1	sum	take	zipWith
p0	.09	.13	.00	.21	.28	.02	.07	.22	.20	.17	.20	.02	.12	.12	.08	.08	.12	.37	.05	.16	.09	.06	.11	.14	.04	.03	.02	.10	.04	.00	.06	.55	.91	.06
p1	.03	.05	.00	.41	.36	.01	.06	.01	.02	.00	.01	.00	.11	.01	.04	.02	.15	.94	.16	.86	.05	.00	.07	1.00	.00	.07	.03	.09	.06	.00	.06	.08	.00	1.00
p2	.33	.47	.00	.43	.36	.00	.05	.98	.07	.06	.05	.02	.52	.01	.38	.30	.01	.30	.31	.01	.30	.01	.04	.00	.00	.07	.07	.10	.07	.00	.06	.01	.05	1.00
p3	.14	.07	.00	.15	.70	.01	.11	.12	.18	.03	.06	.01	.36	.05	.09	.09	.11	.48	.24	.42	.03	.02	.16	.02	.03	.46	.11	.53	.09	.01	.11	.30	.00	.22
p4	.06	.03	.88	.01	.01	.01	.00	.04	.01	.01	.00	.00	.05	.03	.02	.01	.18	.96	.09	.10	.15	.07	.00	.00	.03	.49	.03	.09	.06	.17	.94	.13	.00	.92
p5	.00	.01	.00	.01	.00	.00	.07	.00	.04	.02	.10	.00	.06	.00	.04	.01	.00	.19	.42	.00	.99	.00	.02	.00	.03	.02	1.00	.15	.03	.00	.05	.00	.00	1.00
p6	.12	.08	.00	.68	.40	.08	.09	.12	.11	.08	.07	.01	.22	.10	.08	.09	.23	.47	.23	.02	.28	.53	.03	.00	.00	.09	.07	.41	.08	.00	.06	.17	.01	.99
p7	.03	.02	.97	.26	.27	.00	.08	.00	.12	.05	.01	.00	.17	.05	.03	.02	.08	.75	.07	.02	.22	.01	.37	.00	.04	.03	.06	.57	.09	.02	.14	.92	.00	1.00
p8	.05	.28	.00	.33	.19	.01	.28	.23	.05	.02	.03	.04	.38	.03	.11	.15	.09	.42	.29	.38	.06	.01	.31	.00	.01	.10	.07	.29	.09	.01	.14	.56	.02	.25
priors	.11	.11	.28	.28	.33	.06	.11	.16	.11	.11	.11	.11	.26	.04	.11	.11	.04	.54	.28	.04	.28	.04	.11	.11	.11	.11	.07	.32	.07	.11	.11	.03	.10	.57

Heatmaps

Program	reverse	scanl	sort	square	sub1	sum	take	zipWith
p0	.02	.10	.04	.00	.06	.55	.91	.06
p1	.03	.09	.06	.00	.06	.08	.00	1.00
p2	.07	.10	.07	.00	.06	.01	.05	1.00
p3	.11	.53	.09	.01	.11	.30	.00	.22
p4	.03	.09	.06	.17	.94	.13	.00	.92
p5	1.00	.15	.03	.00	.05	.00	.00	1.00
p6	.07	.41	.08	.00	.06	.17	.01	.99
p7	.06	.57	.09	.02	.14	.92	.00	1.00
p8	.07	.29	.09	.01	.14	.56	.02	.25
priors	.07	.32	.07	.11	.11	.03	.10	.57

P0: Compute the sum of *a* smallest numbers from the list *b*.

Specification:

```
2, [1 5 3]  →  4
1, [1 8 3 5]  →  1
3, [1 8 3 5 7]  →  9
...
```

Target program:

```
a ← int
b ← [int]
c ← Sort b
d ← Take a c
e ← Sum d
```

Heatmaps

Program	reverse	scanl	sort	square	sub1	sum	take	zipWith
p0	.02	.10	.04	.00	.06	.55	.91	.06
p1	.03	.09	.06	.00	.06	.08	.00	1.00
p2	.07	.10	.07	.00	.06	.01	.05	1.00
p3	.11	.53	.09	.01	.11	.30	.00	.22
p4	.03	.09	.06	.17	.94	.13	.00	.92
p5	1.00	.15	.03	.00	.05	.00	.00	1.00
p6	.07	.41	.08	.00	.06	.17	.01	.99
p7	.06	.57	.09	.02	.14	.92	.00	1.00
p8	.07	.29	.09	.01	.14	.56	.02	.25
priors	.07	.32	.07	.11	.11	.03	.10	.57

P1: Count the number of points of the winner. *a* is a list of wins (3 points), and *b* is a list of draws (1 point).

Specification:

```
[1 2], [1 2]        →  8
[1 0 0], [1 1 2]    →  4
[2 2 1 0], [1 1 0 0] →  7
...
```

Target program:

```
a ← [int]
b ← [int]
c ← Map (*3) a
d ← ZipWith (+) c b
e ← Maximum d
```

Heatmaps

Program	reverse	scanl	sort	square	sub1	sum	take	zipWith
p0	.02	.10	.04	.00	.06	.55	.91	.06
p1	.03	.09	.06	.00	.06	.08	.00	1.00
p2	.07	.10	.07	.00	.06	.01	.05	1.00
p3	.11	.53	.09	.01	.11	.30	.00	.22
p4	.03	.09	.06	.17	.94	.13	.00	.92
p5	1.00	.15	.03	.00	.05	.00	.00	1.00
p6	.07	.41	.08	.00	.06	.17	.01	.99
p7	.06	.57	.09	.02	.14	.92	.00	1.00
p8	.07	.29	.09	.01	.14	.56	.02	.25
priors	.07	.32	.07	.11	.11	.03	.10	.57

P4: Compute the minimal total area of rectangles which are constructed by pairing dimensions given in lists *a* and *b*.

Specification:

`[1 2 3], [1 2 3]` $\rightarrow$ 10
`[1 2 2], [1 1 2]` $\rightarrow$ 6
...

Target program:

```
a ← [int]
b ← [int]
c ← Sort a
d ← Sort b
e ← Reverse d
f ← ZipWith (∗) d e
g ← Sum f
```

Outline

Neuro-Guided Genetic Programming

- Fixed-length, linear program representation.
- At the beginning, mutation in GP is parametrized with the result returned by network for the input-output examples in the specification.
- Apart from that, GP proceeds normally.
- All programs in a GP run have the same nominal length, computed as: length of the target program $+ 1$.
- *Nop* operation is included, to allow for effectively shorter programs.

Search operators

Mutation:

- An instruction is randomly selected in the program.
- The function call is analyzed, and constructed is a set of functions with the matching signature.
- A function to insert and its arguments are selected randomly with the probabilities provided by the network (after normalization).

Crossover:

- Exchanging up to $l_c = 2$ consecutive instructions between parents.
- Signatures of the instructions must match.
- If there are no such consecutive instructions, then l_c is decreased.
- If $l_c = 0$, then parent programs are returned unchanged.

Outline

Evolution parameters

- **Preliminary parameter tuning:** the probabilities of mutation and crossover $p_m, p_c \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, population size $\in \{100, 500, 1000\}$; each configuration was ran 30 times.

Parameter	Value
Population size	1000
Max generations	200
Number of runs	50
Probability of mutation p_m	0.8
Probability of crossover p_c	0.0 or 0.5
Selection method	Tournament (**T**) or Lexicase (**L**)
Tournament size	7
Max program length	length of target program + 1
Number of fitness cases	128

Benchmarks

Benchmark	P0	P1	P2	P3	P4	P5	P6	P7	P8
Length	3	3	2	4	5	2	4	3	4
Small training set	✓	✓	✓			✓		✓	
Large training set	✓	✓	✓	✓		✓	✓	✓	✓

Benchmarks the same as in the DeepCoder paper [1].

Experiment dimensions

- ***Small training set*** – 822,582 programs up to length 3.
- ***Large training set*** – 5,004,532 programs up to length 4.

Total tested configurations $= \mathbf{2} \cdot \ldots$

Experiment dimensions

- **T –** Tournament selection (size 7)
- **L –** Lexicase selection

Total tested configurations $= 2 \cdot \mathbf{2} \cdot \ldots$

Experiment dimensions

- **C** – Crossover used ($p_c = 0.5$)
- **N** – Crossover *not* used ($p_c = 0.0$)

Total tested configurations $= 2 \cdot 2 \cdot \mathbf{2} \cdot \ldots$

Experiment dimensions

- **U** – Search operators biased with a *uniform distribution*
- **P** – Search operators biased with *prior probabilities* reflecting the frequency of instructions in the training set
- **S** – Search operators biased using *NN; only mutation*
- **IS** – Search operators biased using *NN; both mutation and population initialization*

Total tested configurations $= 2 \cdot 2 \cdot 2 \cdot \mathbf{4} = 32$

Observation 1

- **IS** is much better than **S**.
- Because of that, in the further analysis we present results only for the **IS** variant.

configuration	avg success rate
S (mut)	0.574
IS (mut, pop_init)	0.735

Observation 2

- Crossover does not make much difference for the effectiveness of search.
- Because of that, in the further analysis we focus on the **N** (no crossover) variant.

configuration	avg success rate
C (crossover)	0.573
N (no crossover)	0.580

Observation 3

- Configurations parametrized with probability estimates were better than baselines.

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

method cx	T_U 0.0	T_P 0.0	T_{IS} 0.0	L_U 0.0	L_P 0.0	L_{IS} 0.0
P2 (2)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P5 (2)	**1.00**	**1.00**	**1.00**	0.98	**1.00**	**1.00**
P0 (3)	0.70	0.34	**1.00**	0.58	0.40	**1.00**
P1 (3)	0.18	0.26	0.54	0.16	0.20	0.96
P7 (3)	0.16	0.34	0.56	**1.00**	**1.00**	**1.00**
P3 (4)	0.14	0.12	**1.00**	0.52	0.28	**1.00**
P6 (4)	0.08	0.06	0.04	0.40	0.82	0.78
P8 (4)	0.18	0.10	0.28	0.36	0.26	0.82
P4 (5)	0.14	0.02	0.00	0.52	0.38	0.14
mean	0.40	0.36	0.60	0.61	0.59	0.86
rank	10.72	12.00	8.28	8.50	8.17	4.17

Observation 3

- Configurations parametrized with probability estimates were better than baselines.

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

method cx	T_U 0.0	T_P 0.0	T_{IS} 0.0	L_U 0.0	L_P 0.0	L_{IS} 0.0
P2 (2)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P5 (2)	**1.00**	**1.00**	**1.00**	0.98	0.98	**1.00**
P0 (3)	0.70	0.34	**1.00**	0.58	0.54	**1.00**
P1 (3)	0.18	0.20	0.58	0.16	0.16	0.98
P7 (3)	0.16	0.28	0.78	**1.00**	0.98	**1.00**
P3 (4)	0.14	0.10	0.68	0.52	0.46	0.98
P6 (4)	0.08	0.00	0.12	0.40	0.64	0.72
P8 (4)	0.18	0.16	0.42	0.36	0.32	0.84
P4 (5)	0.14	0.02	0.00	0.52	0.52	0.32
mean	0.40	0.34	0.62	0.61	0.62	0.87
rank	10.56	12.33	7.28	8.50	9.39	3.56

Observation 4

- Average success rate on the selected benchmarks was slightly higher for the small training set.

training set	avg success rate
small	0.581
large	0.573

Statistical analysis

Ranks for the tested configurations (Friedman's test):

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 ($p = 0.01058$)

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

Legend: small/large (training set used), N (no crossover), C (crossover), T (tournament), L (lexicase), U (unbiased), P (priors baseline), S (search), IS (initialization and search).

Summary

- Neuro-Guided GP first trains the neural network on the family of search problem instances of interest, and then uses this network to guide search.
- Neural network is able to generalize beyond the program size it was trained on.
- Neuro-Guided GP fared better than standard GP and baselines on a small set of problems.

Final words

Thank you for your attention!

Bibliography I

[1] Matej Balog et al. “DeepCoder: Learning to Write Programs”. In: *arXiv preprint arXiv:1611.01989* (2016).

[2] Paweł Liskowski, Iwo Błądek, and Krzysztof Krawiec. “Neuro-guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’18. Kyoto, Japan: ACM, 2018, pp. 1143–1150.