Neurogenetic Programming Framework for Explainable Reinforcement Learning

Vadim Liventsev*
Eindhoven University of Technology
Eindhoven, the Netherlands
v.liventsev@tue.nl

Aki Härmä
Philips Research
Eindhoven, the Netherlands
aki.harma@philips.com

Milan Petković*
Eindhoven University of Technology
Eindhoven, the Netherlands
m.petkovic@tue.nl

ABSTRACT

Automatic programming, the task of generating computer programs compliant with a specification without a human developer, is usually tackled either via genetic programming methods based on mutation and recombination of programs, or via neural language models. We propose a novel method that combines both approaches using a concept of a virtual neuro-genetic programmer, or scrum team. We demonstrate its ability to provide performant and explainable solutions for various OpenAI Gym tasks, as well as inject expert knowledge into the otherwise data-driven search for solutions.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**;
• **Theory of computation** → **Reinforcement learning**;

KEYWORDS

Reinforcement Learning, Program Synthesis, Genetic Programming

ACM Reference Format:
Vadim Liventsev, Aki Härmä, and Milan Petković. 2021. Neurogenetic Programming Framework for Explainable Reinforcement Learning. In *Proceedings of the Genetic and Evolutionary Computation Conference 2021 (GECCO '21).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3449726.3459537

1 INTRODUCTION

Unlike black box machine learning [9], automatic programming promotes exchange of knowledge between human experts and machine learning models:

- Models can generate new programs by applying modifications to expert-written programs, using them as the basis
- Experts can examine the generated programs, understand the algorithm suggested by the system and learn from it

It is usually solved via *genetic programming* [5, 7] where new programs are generated by mutating and mixing a *population* of programs or, more recently, by training *neural models* that generate executable programs as text [2–4]. But "there is no reason whatsoever to not combine both technologies" [1]: a neural network with a finite number of parameters can approximate the space of high-quality programs so that the optimal solution is a few mutations away from the solution sampled from a neural network. In this poster we test this hypothesis and develop a hybrid method. Its open-source implementation can be found at https://github.com/vadim0x60/cibi

2 METHODOLOGY

2.1 Instant Scrum

We call our algorithm *Instant Scrum* in reference to a popular Agile software team work model [13]. It requires

(1) function $Eval(c)$ that runs program c and returns score,
(2) team T of developers $\langle p_{\text{dev}}(c|\theta_{\text{dev}}, C), Update(\theta, c, R)\rangle$ where a developer is defined as a distribution over the space of programs c with trainable parameters θ_{dev} updated with reinforcement learning via function $Update(\theta, c, R)$,
(3) tuple of pre-existing programs C. C is optional in a sense that in can be an empty set, but including them lets the system incorporate expert knowledge,
(4) iteration limit $N_{\max}$

.

Algorithm 1 Instant Scrum with a team of developers

function InstantScrum($Eval, T, C, N_{\max}$)
  for $N \leftarrow 1, \ldots, N_{\max}$ **do**
    for $\langle p_{\text{dev}}, Update\rangle \in T$ **do** ▷ For each developer
      $c_{\text{new}} \sim p_i(c|\theta_i, C)$ ▷ Sample a program
      $R \leftarrow Eval(c_{\text{new}})$ ▷ Test it
      $C \leftarrow C \cup \{\langle c_{\text{new}}, R\rangle\}$ ▷ Save to the codebase
      $\theta_i \leftarrow Update(\theta_i, c, R)$ ▷ Train the developer
    end for
  end for
  return $\arg\max_{c_{\text{uniq}}} \left(\sum_{\langle c,R\rangle}^{C} \mathbb{I}[c_{\text{uniq}} = c]R \right) / \left(\sum_{\langle c,R\rangle}^{C} \mathbb{I}[c_{\text{uniq}} = c] \right)$
end function

This method enables several heterogeneous program synthesis methods to contribute to a common task and learn from each other. *Neurogenetic Programming* is *Instant Scrum* with team T that includes both genetic and neural developers (described below)

2.2 Genetic developer

2.2.1 Sampling procedure. Genetic developer contributes to codebase C by sampling 2 programs c_1 and c_2 from C and merging them with one of 7 genetic operators. Programs are sampled from the following distribution:

*Also with Philips Research.

GECCO '21, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3459537

Parent 1	ae>>>>>34+
Parent 2	a[e>-a-]b[e>>-b-]
Shuffle mutation	>>4+>3>e>a
Uniform mutation	ae@>!>>35+
1-point crossover	ae>>>>-]b[e>>-b-]
2-point crossover	ae>>-a-34+
Uniform crossover	aee>->>3b+
Messy crossover	ae>>>>e>-a-]b[e>>-b-]
Pruning	e>>>>>4+

Table 1: All operators applied a pair of BF++ [10] programs

$$p(c_{\text{sampled}}|C) \sim \frac{\sum\limits_{\langle c,R\rangle}^{C} \mathbb{I}[c_{\text{sampled}} = c]e^R}{\sum\limits_{\langle c,R\rangle}^{C} \mathbb{I}[c_{\text{sampled}} = c]} \tag{1}$$

That is, proportional to exponentiated score, averaged over all copies of the program in C. Such weighting reduces variance by averaging over several attempts to evaluate the program. It also (due to exponentiation) prioritizes programs that have been very successful a few times, even if their average R is low. We consider such programs to be *high-quality additions to the codebase* because they contain the knowledge necessary for succeeding, even if on average they don't.

2.2.2 Operators. Once c_1 and c_2 are chosen, genetic developer picks one of available operators [5, 6] (see table 1): output a shuffled version of c_1; replace some tokens with a random one from the alphabet; merge c_1 and c_2 with one- or two-point or messy crossover - an operator inspired by DNA [11] that randomly picks a breakpoints in c_1 and c_2 and copies characters before the breakpoint from c_1, the rest from c_2; uniform crossover (tossing a coin for each character to decide whether it will come from c_1 or c_2)

After initial experiments we found that generated programs often contain sections of unreachable code or code that makes changes to the execution state and fully reverses them. To address this, we introduced an additional operator for removing dead code (*pruning*): pruning helps separate sections of this program that led to its success from sections that appeared in a highly-rated program by accident.

The task of choosing a genetic operator to generate a program with high R is a *multi-armed bandit problem* [12] solved using one of the standard bandit algorithms [8], such as *epsilon-greedy optimization*.

2.3 Neural developer

The *neural developer*, also known as the *senior developer* because of their unique ability to write original programs, is an LSTM network followed by a linear layer that generates a sequence of vectors $h_1, h_2, h_3, \ldots$ where $h_i \in \mathbb{R}^{|\mathcal{L}|+1} \forall i$ and j-th element of vector h_i, $h_i^{(j)}$, represents the probability of i-th token of the program being j-th token in the alphabet, $p(c^{(i)} = \mathcal{L}^{(j)})$. The last element of the vector represents a special *end of program* symbol.

For the $Update_{\text{neural}}$ procedure we use the algorithm proposed in [2]. The subproblem of generating a program c is considered as a reinforcement learning episode of it's own, where tokens are actions and token number $|c|+1$ (*end of program* token) is assigned reward $q = e^R; R \sim Eval(c)$. In this subenvironment $h_i(\theta)$ is the policy network [14, chapter 13] trained using REINFORCE algorithm with Priority Queue Training. This algorithm involves a priority queue of best known programs. In our case, priority queue is best programs from C as weighted by (1) which means that the neural developer can train on programs written by other developers.

$h_i(\theta)$ can also represent several LSTM layers stacked or a different type of recurrent neural network, i.e. GRU.

2.4 Dummy developer

The last type of developer in T is the simplest one: it samples a program from distribiton (1) and outputs its identical copy. The reason for this is to get the program evaluated again so that the average R of this program better approximates its expected score. Adding a dummy developer helps avoid programs that achieve a high score by accident once and derail the entire process.

ACKNOWLEDGMENTS

This work was funded by the European Union's Horizon 2020 research and innovation programme under grant agreement 812882. This work is part of "Personal Health Interfaces Leveraging HUman-MAchine Natural interactionS" (PhilHumans) project

REFERENCES

[1] [n. d.]. Algorithm Synthesis: Deep Learning and Genetic Programming. http://iao.hfuu.edu.cn/blogs/33-algorithm-synthesis-deep-learning-and-genetic-programming. ([n. d.]). (Accessed on 02/03/2021).
[2] Daniel A. Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V. Le. 2018. Neural Program Synthesis with Priority Queue Training. *arXiv preprint arXiv:1801.03526* (2018). arXiv:1801.03526 http://arxiv.org/abs/1801.03526
[3] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*. PMLR, 245–256.
[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989
[5] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming: an introduction*. Vol. 1. Morgan Kaufmann Publishers San Francisco.
[6] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.
[7] John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.
[8] Volodymyr Kuleshov and Doina Precup. 2014. Algorithms for multi-armed bandit problems. *CoRR* abs/1402.6028 (2014). arXiv:1402.6028 http://arxiv.org/abs/1402.6028
[9] Yuxi Li. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).
[10] Vadim Liventsev, Aki Härmä, and Milan Petković. 2021. BF++:a language for general-purpose neural program synthesis. (2021). arXiv:cs.AI/2101.09571
[11] Thomas Hunt Morgan. 1916. *A Critique of the Theory of Evolution*. Princeton University Press.
[12] Herbert Robbins. 1952. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.* 58, 5 (1952), 527–535.
[13] Ken Schwaber and Mike Beedle. 2002. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River.
[14] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.