Genetic Programming and Evolvable Machines (2019) 20:139–142
https://doi.org/10.1007/s10710-018-9341-4

SOFTWARE REVIEW

Software review: DEAP (Distributed Evolutionary Algorithm in Python) library

Jinhan Kim[1] · Shin Yoo[1]

Published online: 21 November 2018

Abstract
We give a critical assessment of the DEAP (Distributed Evolutionary Algorithm in Python) open-source library and highly recommend it to both beginners and experts alike. DEAP supports a range of evolutionary algorithms including both strongly and loosely typed Genetic Programming, Genetic Algorithm, and Multi-Objective Evolutionary Algorithms such as NSGA-II and SPEA2. It contains most of the basic functions required by evolutionary computation, so that its users can easily construct various flavours of both single and multi-objective evolutionary algorithms and execute them using multiple processors. It is ideal for fast prototyping and can be used with an abundance of other Python libraries for data processing as well as other machine learning techniques.

1 Introduction

DEAP is built on Python and designed for both practitioners and researchers who need fast prototyping or technical experiments using evolutionary computation. It was introduced at GECCO 2012 [2], and has been widely used by researchers from many different backgrounds [1, 7, 9, 11]. Its design aim is to provide users with appropriate tools so that they can easily write their own evolutionary loops, while supporting parallelism transparently. The latest release version of DEAP is 1.2.2, and supports both Python 2 and 3. DEAP is maintained by Computer Vision and Systems Laboratory (CVSL) at Laval University, Canada, and is available from https://github.com/DEAP/deap.

Jinhan Kim
jinhankim@kaist.ac.kr

Shin Yoo
shin.yoo@kaist.ac.kr

[1] School of Computing, KAIST, 291 Daehak Ro, Yuseong Gu, Daejeon 34141, Republic of Korea

```
 1 from deap import creator
 2 from deap import gp
 3 ...
 4
 5 creator.create("FitnessMax", base.Fitness, weights=(1.0,))
 6 creator.create("Individual", gp.PrimitiveTree, fitness=creator.
       FitnessMax)
 7
 8 toolbox = base.Toolbox()
 9 toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1,
       max_=2)
10 toolbox.register("individual", tools.initIterate, creator.
       Individual, toolbox.expr)
11 toolbox.register("population", tools.initRepeat, list, toolbox.
       individual)
12 toolbox.register("compile", gp.compile, pset=pset)
13 toolbox.register("select", tools.selTournament, tournsize=3)
14 toolbox.register("mate", gp.cxOnePoint)
15 toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
16 toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut,
       pset=pset)
17
18 toolbox.register("evaluate", myEval, args1=arg_list1, arg2=
       arg_list2)
19
20 toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("
       height"), max_value=10))
21 toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter(
       "height"), max_value=10))
22 ...
```

Fig. 1 An example evolutionary loop written in DEAP

2 Major Features

Written in Python, DEAP can be run on Linux, Microsoft Windows, and Mac OS X. It requires Python 2.6 or above. By default, DEAP supports array-like representations as well as the tree-based representation that enables tree GP. Users can configure and execute Genetic Programming in an intuitive and declarative manner using DEAP. At its core, DEAP implements its evolutionary algorithm using pluggable genetic and selection operators. Users can create various flavours of Genetic Programming and Genetic Algorithms by combining different parts at the API level. Many widely used genetic operators are already implemented within DEAP. Consider the code snippet in Fig. 1, which sets up a single objective (Line 5) tree GP (Line 6) with half and half ramping initialisation (Line 9), three-way tournament selection (Line 13), single point crossover (Line 14) and subtree regeneration mutation of maximum depth 2 (Line 15). The fitness function is a user-defined Python function called myEval (Line 18), which can accept arbitrary arguments. Finally, both crossover and mutation are constrained by tree height of 10 (Lines 20 and 21).

3 Weaknesses

Let us discuss what we think are weaknesses of DEAP. First, a basic knowledge of Python is essential. DEAP is a programming library, and running GP with DEAP means writing a Python program, however simple it may be. Therefore, for beginners, the developers of DEAP have provided various working examples. Second, there is no Graphical User Interface (GUI) and support for graphical plots is rather weak. While the lack of a GUI is expected as a programming library, DEAP does not support graphical plotting of experimental results, such as found in jMetal [3]. The only type of plot supported by DEAP is individual geneology.[1] Finally, the programming language itself has performance limitations. Python is an interpreted language and, is on average, much slower than compiled languages, such as C. For applications with expensive fitness computation, this may be a significant weakness.

4 Strengths

Despite the weaknesses, we highly recommend DEAP not only as a GP library but also for other evolutionary algorithms, regardless of the level of GP expertise the user has. For those who wants to use Genetic Programming for the first time, DEAP provides all the basic data structures, genetic operators, and basic examples, so that the user can quickly implement an evolutionary loop. If the user is already familiar with GP, DEAP is sufficiently flexible to allow more advanced variants, such as fitness evaluation using parallel graphics hardware, GPGPU [5, 6].

We also consider the fact that DEAP is written in Python as a benefit, as it is arguably one of the most popular languages for data scientists. There are a wide variety of data processing and other machine learning algorithm libraries that are written in Python (for example, data processing library `pandas` [8] and general machine learning library `scikit-learn` [10]): we think the interoperability through the shared programming language will help those who have never experienced GP before to adopt it as a learning technique more easily. The benefits of Python go beyond the popularity of the language. Python is an interpreted language that also allows on-the-fly compilation of textual code to Python bytecode: for GP, this means that non-terminal GP node operators can be just Python functions, and fitness evaluation can be execution of *compiled* individual in the format of function calls.

Finally, as its name suggests, DEAP supports various levels of parallelism ranging from Python's `multiprocessing` to distributed grid computing. This is achieved via a mature concurrent computation framework called SCOOP (Scalable Concurrent Operations in Python) [4].

[1] http://deap.readthedocs.io/en/master/api/tools.html#history.

5 Conclusions

DEAP is a well designed GP library for novices and experts alike. It is easy to use, ideal for both fast prototyping and grid-computing level distributed experiments. The fact that it is written in Python will not only allow easier adoption for many researchers and practitioners but also mean that it can be used with many other Python libraries for data processing and machine learning.

Acknowledgements This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MEST) (Grant No. NRF-2016R1C1B1011042).

References

1. S. Chardon, B. Brangeon, E. Bozonnet, C. Inard, Construction cost and energy performance of single family houses: from integrated design to automated optimization. Autom. Constr. **70**, 1–13 (2016)
2. F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, DEAP Christian Gagné. DEAP: A python framework for evolutionary algorithms, in Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO’12, Philadelphia, Pennsylvania, USA (ACM, 2012), pp. 85–92
3. J.J. Durillo, A.J. Nebro, jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)
4. Y. Hold-Geoffroy, O. Gagnon, M. Parizeau, Once you scoop, no need to fork, in Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, (ACM, 2014), p. 60
5. J. Kim, J. Kim, S. Yoo, GGPGPGPU: evaluation of parallelisation of genetic programming using GPGPU, in International Symposium on Search-Based Software Engineering, SSBSE 2017, (Springer, 2017), pp. 137–142
6. W.B. Langdon, W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards, in M. O’Neill, L. Vanneschi, S. Gustafson, A.I.E. Alcazar, I. De Falco, A.D. Cioppa, E. Tarantino, editors, Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, Volume 4971 of Lecture Notes in Computer Science, Naples, 26–28 March 2008. Springer, pp. 73–85
7. MMJ Macret, Automatic tuning of the OP-1 synthesizer using a multi-objective genetic algorithm. Master’s thesis, Communication, Art & Technology: School of Interactive Arts and Technology, Simon Fraser University, Vancouver, Canada, 16 July 2013
8. W. McKinney, Data structures for statistical computing in python, in S. van der Walt, J. Millman, editors, Proceedings of the 9th Python in Science Conference, pp. 51–56 (2010)
9. R.S. Olson, R.J. Urbanowicz, P.C. Andrews, N.A. Lavender, L.C. Kidd, J.H. Moore. Automating biomedical data science through tree-based pipeline optimization, in European Conference on the Applications of Evolutionary Computation, Volume 9597 of Lecture Notes in Computer Science, (Springer, 2016), pp. 123–137
10. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
11. J. Sohn, S. Yoo, FLUCCS: Using code and change metrics to improve fault localisation. Proceedings of the International Symposium on Software Testing and Analysis, ISSTA **2017**, 273–283 (2017)