Journal of Machine Learning Research 13 (2012) 2171-2175 Submitted 7/11; Revised 4/12; Published 7/12

DEAP: Evolutionary Algorithms Made Easy

Félix-Antoine Fortin FELIX-ANTOINE.FORTIN.1@ULAVAL.CA
François-Michel De Rainville FRANCOIS-MICHEL.DE-RAINVILLE.1@ULAVAL.CA
Marc-André Gardner MARC-ANDRE.GARDNER.1@ULAVAL.CA
Marc Parizeau MARC.PARIZEAU@GEL.ULAVAL.CA
Christian Gagné CHRISTIAN.GAGNE@GEL.ULAVAL.CA
Laboratoire de vision et systèmes numériques
Département de génie électrique et de génie informatique
Université Laval
Québec (Québec), Canada G1V 0A6

Editor: Mikio Braun

Abstract

DEAP is a novel evolutionary computation framework for rapid prototyping and testing of ideas. Its design departs from most other existing frameworks in that it seeks to make algorithms explicit and data structures transparent, as opposed to the more common black-box frameworks. Freely available with extensive documentation at `http://deap.gel.ulaval.ca`, DEAP is an open source project under an LGPL license.

Keywords: distributed evolutionary algorithms, software tools

1. Introduction

Evolutionary Computation (EC) is a sophisticated field with very diverse techniques and mechanisms, where even well designed frameworks can become quite complicated under the hood. They thus strive to hide the implementation details as much as possible, by providing large libraries of high-level functionalities, often in many different flavours. This is the black-box software model (Roberts and Johnson, 1997). The more elaborate these boxes become, the more obscure they are, and the less likely the commoner is to ever take a peek under the hood to consider making changes. But using EC to solve real-world problems most often requires the customization of algorithms.

The DEAP (Distributed Evolutionary Algorithms in Python) framework is built over the Python programming language that provides the essential glue for assembling sophisticated EC systems. Its aim is to provide practical tools for rapid prototyping of custom evolutionary algorithms, where every step of the process is as explicit (pseudocode like) and easy to read and understand as possible. It also places a high value on both code compactness and code clarity.

2. Core Architecture

DEAP's core is composed of two simple structures: a creator and a toolbox. The *creator* module is a meta-factory that allows the run-time creation of classes via both inheritance and composition. Attributes, both data and functions, can be dynamically added to existing classes in order to cre-

ate new types empowered with user-specific EC functionalities. Practically speaking, this allows the creation of genotypes and populations from any data structure such as lists, sets, dictionaries, trees, etc. This creator concept is key to facilitating the implementation of any type of EA, including genetic algorithms (Mitchell, 1998), genetic programming (Banzhaf et al., 1998), evolution strategies (Beyer and Schwefel, 2002), covariance matrix adaptation evolution strategy (Hansen and Ostermeier, 2001), particle swarm optimization (Kennedy and Eberhart, 2001), and many more.

The *toolbox* is a container for the tools (operators) that the user wants to use in his EA. The toolbox is manually populated by the user with selected tools. For instance, if the user needs a crossover in his algorithm, but has access to several crossover types, he will choose the one best suited for his current problem, for example a uniform crossover "cxUniform", and register it into the toolbox using a generic "mate" alias. This way, he is able to build algorithms that are decoupled from operator sets. If he later decides that some other crossover is better suited, his algorithm will remain unchanged, he will only need to update the corresponding alias in the toolbox.

The core functionalities of DEAP are levered by several peripheral modules. The *algorithms* module contains four classical EC algorithms: generational, $(\mu\ ,\ \lambda)$, $(\mu+\lambda)$, and ask-and-tell (Collette et al., 2010). These serve as a starting point for users to build their own custom EAs meeting their specific needs. The *tools* module provides basic EC operators such as initializations, mutations, crossovers, and selections. These operators can be directly added to a toolbox in order to be used in algorithms. This module also contains a number of components that gather useful information regarding the evolution: fitness statistics, genealogy, hall-of-fame for the best individuals encountered so far, and checkpointing mechanisms to allow evolution restart. A *base* module contains some data structures frequently used in EC, but not implemented in standard Python (e.g., generic fitness object). The last module named *dtm*, for Distributed Task Manager, offers distributed substitutes for common Python functions such as `apply` and `map`. DTM allows distribution of specific parts of users' algorithms by taking care of spawning and distributing sub-tasks across a cluster of computers. It even balances the workload among workers to optimize the distribution efficiency.

To illustrate DEAP usage, we now present an example for **multi-objective feature selection**. The individual is represented as a bit-string where each bit corresponds to a feature that can be selected or not. The objective is to maximize the number of well-classified test cases and to minimize the number of features used. Figure 1 shows how this problem can be solved using DEAP.

On line 2, the relevant DEAP modules are first imported. On line 3, a multi-objective fitness class `FitnessMulti` is created. The first argument of the `creator.create` method defines the name of the derived class, while the second argument specifies the inherited base class (in this case `base.Fitness`). The third argument adds a new class attribute called `weights`, initialized with a tuple that specifies a two-objective fitness, of which the first must be maximized (1.0), and the second must be minimized (-1.0). Next, an `Individual` class is derived from the Python list and composed with our newly created `FitnessMulti` object. After defining a proper evaluation function that returns the fitness values (classification rate, number of selected features) (lines 6 and 7), a toolbox object is created on line 9, and populated on lines 10 through 16 with aliases in order to initialize individuals and population, and specify the variation operators (`mate`, `mutate`, and `select`) and the fitness evaluation function (`evaluate`) used by the evolutionary loop. The toolbox `register` method accepts a variable number of arguments. The first is the alias name, and the second the function that we want to associate with this alias. All other arguments are passed to this function when the alias is called. For instance, a call to `toolbox.bit` will in fact call `random.randint` with arguments 0 and 1, and thus generate a random bit. On line 11, to initialize an individual assuming a 80 features selec-

```
1  import knn, random
2  from deap import creator, base, tools, algorithms
3  creator.create("FitnessMulti", base.Fitness, weights=(1.0, -1.0))
4  creator.create("Individual", list, fitness=creator.FitnessMulti)
5
6  def evalFitness(individual):
7      return knn.classification_rate(features=individual), sum(individual)
8
9  toolbox = base.Toolbox()
10 toolbox.register("bit", random.randint, 0, 1)
11 toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.bit, n=80)
12 toolbox.register("population", tools.initRepeat, list, toolbox.individual, n=100)
13 toolbox.register("evaluate", evalFitness)
14 toolbox.register("mate", tools.cxUniform, indpb=0.1)
15 toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
16 toolbox.register("select", tools.selNSGA2)
17
18 population = toolbox.population()
19 fits = toolbox.map(toolbox.evaluate, population)
20 for fit, ind in zip(fits, population):
21     ind.fitness.values = fit
22
23 for gen in range(50):
24     offspring = algorithms.varOr(population, toolbox, lambda_=100, cxpb=0.5, mutpb=0.1)
25     fits = toolbox.map(toolbox.evaluate, offspring)
26     for fit, ind in zip(fits, offspring):
27         ind.fitness.values = fit
28     population = toolbox.select(offspring + population, k=100)
```

Figure 1: Multi-objective feature selection example with NSGA-II (Deb et al., 2002).

tion problem, this bit function is simply called $n = 80$ times repeatedly using the `tools.initRepeat` method that accepts three arguments: a container, a function, and the number of times to repeat initialization. Similarly, a `population` alias is defined to allow population initialization, in this case with $n = 100$ individuals. Line 18 then proceeds with the allocation of a population of individuals that have their fitness evaluated on line 19 by mapping the evaluation function to every element of the population container. Lines 20 and 21 replace the individuals fitness with their newly computed values. Finally, lines 23 through 28 show the evolutionary loop that uses the $\mu + \lambda$ strategy, where $\mu = 100$ parents (current population) are mixed with $\lambda = 100$ offspring (`lambda_` line 24) for the selection process (NSGA-II) to produce the next generation of $k = 100$ parents (line 28). The `varOr` algorithm loops until it has generated λ offspring using either crossover (`mate` alias) with probability `cxpb`, mutation (`mutate` alias) with probability `mutpb`, or reproduction.

Efficient **distribution** of specific parts of user algorithms is made possible by the `dtm` module. For instance, in the previous examples, the only required changes in order to distribute the fitness evaluation task on a cluster of computers are to import the `dtm` module and to replace the toolbox default `map` function by its distributed `dtm` equivalent:

```
from deap import dtm
toolbox.register("map", dtm.map)
```

The map function calls (lines 19 and 25 in Figure 1) spawn sub-tasks of the evaluation function. The task manager distributes these sub-tasks across the worker nodes, and automatically balances the workload evenly among them.

Framework	Type	Configuration	Algorithm	Example	Total
ECJ	202	35	65	26	328
EO	43	n/a	67	68	178
Open BEAGLE	256	41	116	64	477
Pyevolve	42	n/a	336	25	378
inspyred	23	n/a	143	24	190
DEAP	n/a	n/a	n/a	59	59

Table 1: Comparison of the number of lines needed to define different components of a OneMax example with some major frameworks, as counted by cloc (`http://cloc.sf.net`).

Table 1 presents a comparison of the number of lines required by some of the most popular frameworks to define different components of a OneMax example (without distribution). Columns respectively indicate how many lines are required to define the *bit-string* type, to configure the operators, to implement the generational algorithm and to execute the example. DEAP is the only framework that allows the complete definition of the EA in less than one hundred lines of code. Even though some frameworks may allow shorter expressions of standardized solutions, any needed customization of type or algorithm can also force the user to dig deep into the framework to modify perhaps hundreds of lines. We therefore assert that DEAP is superior to previous frameworks for rapid prototyping of new algorithms and definition of custom types.

3. Conclusion

Current major EC frameworks generally do a good job of offering generic tools to solve hard problems using EAs. However, their implementation intricacies can also make them difficult to extend for the commoner. Even experts can become overwhelmed when trying to implement specific features. This paper introduced the DEAP framework that combines the flexibility and power of the Python programming language with a clean and lean core of transparent EC components that both facilitate rapid prototyping and testing of new EA ideas, and encourage creativeness through simplicity and explicit algorithms. The DEAP core is implemented in 6 program files (Python modules) with only 1653 lines of code. To these core lines, the DTM module adds another 2073 lines for enabling the easy distribution of computationally intensive algorithm parts. The framework also includes more than 35 application examples that add another 2448 lines of code, leading to an average ratio of core lines to example lines of 1.5, compared with 2.4 for inspyred, 3.4 for ECJ, 5.4 for Pyevolve, 5.5 for EO, and 16.3 for OpenBEAGLE. In other words, DEAP implements the 15 examples of OpenBEAGLE plus 20 more with 10 times less lines of code.

Acknowledgments

This work has been supported by the Fonds de recherche du Québec - Nature et technologies (FQRNT) and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.

H.-G. Beyer and H.-P. Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing*, 1:3–52, 2002.

Y. Collette, N. Hansen, G. Pujol, D. Salazar Aponte, and R. Le Riche. On object-oriented programming of optimizers – Examples in Scilab. In P. Breitkopf and R. F. Coelho, editors, *Multidisciplinary Design Optimization in Computational Mechanics*, chapter 14, pages 527–565. Wiley, 2010.

K. Deb, A. Pratab, S. Agarwal, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:849–858, April 2002.

N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, 2001.

M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.