The Journal of Open Source Software

dcgp: Differentiable Cartesian Genetic Programming made easy.

Dario Izzo[1] **and Francesco Biscani**[2]

1 Advanced Concepts Team, European Space Research and Technology Center (Noordwijk, NL) **2** Max Planck Institute for Astronomy (Heidelberg, D)

DOI: 10.21105/joss.02290

Software

- Review
- Repository
- Archive

Editor: Viviane Pons

Reviewers:

- @Ohjeah
- @shah314

Submitted: 21 May 2020
Published: 16 July 2020

Summary

Genetic Programming (GP), is a computational technique based on the idea of representing a computer program in some dedicated data-structure (e.g. a tree) to then evolve it using genetic algorithms as to improve its ability at solving a predefined task. How good a certain program is at solving the task is typically encapsulated in its *fitness*: a vector of floating point values defining various aspects of the program. In a typical example, the *fitness* would be the program output error with respect to a predefined behaviour over a number of inputs. Generation after generation, the genetic algorithm would then improve such a fitness in the attempt to have the program behave correctly.

The idea flourished in the late 80s mainly thanks to the work of John Koza (Koza, 2010), and (using the words of Koza) has always had a "hidden skeleton in the closet": the difficulty to find and evolve real valued parameters in the expressed program. A recent development called **differentiable genetic programming** (Izzo, Biscani, & Mereta, 2017), was introduced to address exactly this issue by allowing to learn constants in computer programs using the differential information of the program outputs as obtained using automated differentiation techniques.

The evolution of a **differentiable genetic program** can be supported by using the information on the derivatives (at any order) of the program outputs with respect to chosen parameters, enabling in GP the equivalent of back-propagation in Artificial Neural Networks (ANN). The fitness of a program can thus be defined also in terms of its derivatives, allowing to go beyond symbolic regression tasks and, for example, to solve differential equations, learn differential models, capture conserved quantities in dynamical systems, search for Lyapunov functions in controlled systems, etc..

In this work we introduce the C++ library `dcgp` and the Python package `dcgpy`, tools we developed to allow research into the applications enabled by **differentiable genetic programming**. In the rest of this paper, we will refer mainly to `dcgp`, with the understanding that the a corresponding behaviour can always be also obtained in `dcgpy`.

A different library called CGP-Library (Turner & Miller, 2015) is already available as a cross platform C implementation of Cartesian Genetic Programming, designed to be simple to use and extendable. The `dcgp` library here introduced, differs in several aspects: it allows to perform derivatives on the represented programs (hence implementing a differentiable form of genetic programming), it allows to use Python to use runtime scripting, it is thread-safe and it allows to define kernels as generic functors.

Methods

In dcgp computer programs are encoded using the Cartesian Genetic Programming (CGP) encoding (Miller, 2011), that is an acyclic graph representation. A Cartesian genetic program, in its original form, is depicted in Figure 1 and is defined by the number of inputs n, the number of outputs m, the number of rows r, the number of columns c, the levels-back l, the arity a of its functional units (*kernels*) and the set of possible *kernels*. With reference to Figure 1, each of the $n + rc$ nodes in a CGP is assigned a unique id. The vector of integers:

$$\mathbf{x}_I = [F_0, C_{0,0}, C_{0,1}, ..., C_{0,a}, F_1, C_{1,0}, ..., O_1, O_2, ..., O_m]$$

defines entirely the value of the terminal nodes and thus the computer program.

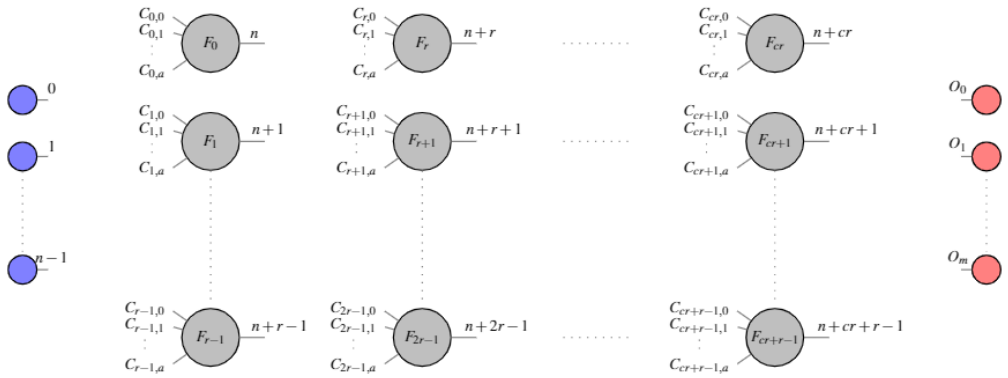

Figure 1: A classical CGP.

Several *kernels* are already available in dcgp, but the user can define his own both in the C++ and in the Python version, allowing for the possibility to consider kernels that are, themselves, genetic programs.

In dcgp the CGP representations of programs are all derived from a base C++ templated class `dcgp::expression<T>`. The use of the template parameter `T` allows to compute the nodes of the acyclic graph defining the computer program, and thus its outputs, using different computational types. In particular, the use of **generalized dual numbers**, implemented as a new type in the library `audi` (Izzo & Biscani, 2020), are enabled and can be used to obtain the derivatives, at any order, of the program outputs with respect to parameters present in its encoding. **Generalized dual numbers** are elements of the algebra of truncated Taylor polynomials and act, in this context, as a high order, forward mode, automated differentiation method. The library `audi` makes computations in this algebra rather straight-forward by defining all the basic arithmetics rules for the new type `audi::gdual<T>` which, essentially, represents a multivariate Taylor polynomial and thus contains all the derivatives of the considered expansion.

The two classes `dcgp::expression_weighted<T>` and `dcgp::expression_ann` derive from the base class `dcgp::expression<T>` and offer new, extended, kinds of CGP representations. `dcgp::expression_weighted<T>` adds a weight to each node connection and thus creates a program rich in floating point constants to be learned. `dcgp::expression_ann` adds also a bias to each node, thus making it possible to represent generic artificial neural networks. Since forward mode automated differentiation is highly unefficient whenever a high number of parameters are to be learned (a typical situation when training ANN weights and biases), `dcgp::expression_ann` is only allowed to operate on the type `double`. Its weights and biases can be learned using backward mode automated differentiation (back propagation) and a stochastic gradient descent algorithm implemented specifically for the class.

All computer programs represented in `dcgp` can be *mutated* via the corresponding methods of the `dcgp::expression<T>` base class, and thus evolved. Mutations can be chosen selectively to affect only specific part of the chromosome, thus affecting only the *kernels*, the connections or the output nodes. As an example, the different consequences of such mutations are visualized, in the case of a `dcgp::expression_ann`, in Figure 2 and studied preliminarily in (Märtens & Izzo, 2019). No crossover is implemented at the level of the `dcgp::expression<T>`, as the original technique employed to evolve Cartesian genetic programs (Miller, 2011) was purely mutation based. We do intend to add this possibility in future versions.

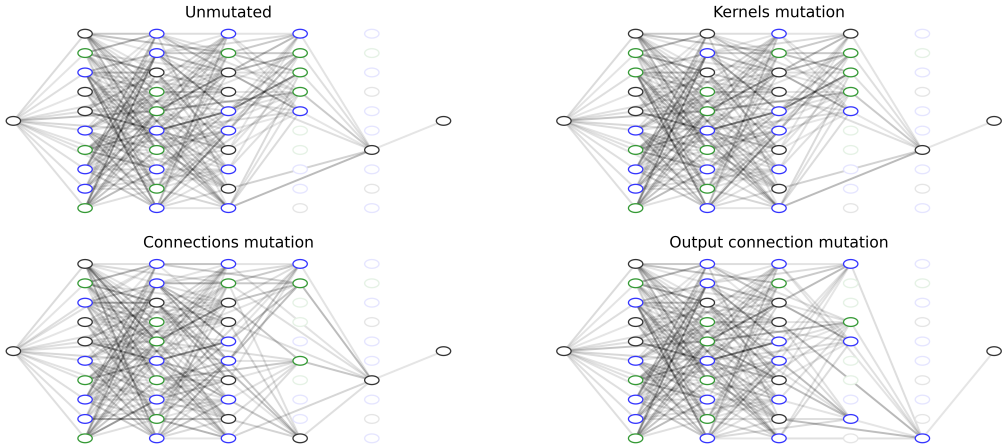

Figure 2: Effects of mutations on a dCGPANN.

When using a `dcgp::expression<T>` for a symbolic regression task, `dcgp` includes also several novel evolutionary startegies (Schwefel, 1993), **memetic** and **multiobjective**, as well as a dedicated `dcgp::symbolic_regression` class that represents such problems as an optimization problem.

Notes on efficiency

The process of evolving a CGP can be lengthy and, being subject to a degree of randomness, is typically repeated a few times to explore the diversity of possible evolved solutions. As a consequence, CPU efficiency is an enabler to build a successful learning pipeline. In the C++ library `dcgp` there are several parallelization levels that can be exploited for this purpose. To start with, the program loss with respect to its expected output, can be parallelized when computed on multiple points. A second layer of parallelization is offered by `audi` (Izzo & Biscani, 2020) when **generalized dual numbers** are employed. In this case, the underlying truncated Taylor polynomial algebra makes use of fine grained parallelization. A third layer is also present in the non **memetic** optimization algorithms shipped with `dcgp` as they can evaluate the fitness of the various individuals in parallel using a batch fitness evaluator. All the resulting nested parallelism is dealt with using the ®Intel threading building blocks library. In the Python package `dcgpy` things are different as multi-threading is not permitted by the Python interpreter and the multi-process paradigm is highly unefficient for a fine grained parallelization. As a consequence, most of the mentioned parallelization are unavailable. The use of coarse-grained parallelization paradigms, such as the island model for evolutionary algorithms, is thus suggested to achieve the best out of `dcgpy` computational times.

C++ and Python APIs

Two separate APIs are maintained for the C++ and Python version of the code. While the APIs are designed to be as close as possible, and to offer a similar experience, there are inevitable differences stemming from the very different capabilities of the languages (to mention one, think about the lack of an equivalent to C++ templates in Python).

In Python a typical initial use of the dcgpy package, would look like:

```
import dcgpy
ks = dcgpy.kernel_set_double(["sum", "diff", "div", "mul"])
ex = dcgpy.expression_double(inputs = 1, outputs = 1, rows = 1,
                cols = 6, levels_back = 6, arity  = 2, kernels = ks())
print("Numerical output: ", ex([1.2]))
print("Symbolic output: ", ex(["x"]))
ex.mutate_active(2)
print("Numerical output: ", ex([1.2]))
print("Symbolic output: ", ex(["x"]))
```

and produce the output:

```
Numerical output:  [1.7]
Symbolic output:  ['((x/(x+x))+x)']
Numerical output:  [1.68]
Symbolic output:  ['((x*(x+x))-x)']
```

while in C++ a typical use of the dcgp library achieving a similar result would look like:

```
#include <dcgp.hpp>
int main() {
    dcgp::kernel_set<double> ks({"sum", "diff", "div", "mul"});
    dcgp::expression<double> ex(1u, 1u, 1u, 6u, 6u, 2u, ks());
    std::cout << "Numerical output: ", ex({1.2})) << "\n";
    std::cout << "Symbolic output: ", ex({"x"})) << "\n";
    ex.mutate_active(2);
    std::cout << "Numerical output: ", ex({1.2})) << "\n";
    std::cout << "Symbolic output: ", ex({"x"})) << "\n";
}
```

Acknowledgments

We acknowledge the important support of Luca Guj and Dow Corporation during the development of the API and documentation.

References

Izzo, D., & Biscani, F. (2020). AuDi: V1.8. *Zenodo*. doi:10.5281/zenodo.3702783

Izzo, D., Biscani, F., & Mereta, A. (2017). Differentiable genetic programming. In *European conference on genetic programming* (pp. 35–51). Springer. doi:10.1007/978-3-319-55696-3_3

Koza, J. R. (2010). Human-competitive results produced by genetic programming. *Genetic programming and evolvable machines*, *11*(3-4), 251–284. doi:10.1007/s10710-010-9112-3

Märtens, M., & Izzo, D. (2019). Neural network architecture search with differentiable cartesian genetic programming for regression. In *Proceedings of the genetic and evolutionary computation conference companion* (pp. 181–182). Association for Computing Machinery (ACM). doi:10.1145/3319619.3322003

Miller, J. F. (2011). Cartesian genetic programming. In *Cartesian genetic programming* (pp. 17–34). Springer. doi:10.1007/978-3-642-17310-3_2

Schwefel, H.-P. P. (1993). *Evolution and optimum seeking: The sixth generation*. John Wiley & Sons, Inc. ISBN: 978-0-471-57148-3

Turner, A. J., & Miller, J. F. (2015). Introducing a cross platform open source cartesian genetic programming library. *Genetic Programming and Evolvable Machines*, *16*(1), 83–91. doi:10.1007/s10710-014-9233-1