The Journal of Open Source Software

GAIM: A C++ library for Genetic Algorithms and Island Models

Georgios Detorakis[1, 2] **and Andrew Burton**[1, 2]

1 Department of Cognitive Sciences, University of California Irvine, Irvine, USA. **2** All authors contributed equally to this work.

DOI: 10.21105/joss.01839

Software

- Review
- Repository
- Archive

Editor: Mark A. Jensen

Reviewers:

- @sjvrijn
- @sarats

Submitted: 15 October 2019
Published: 02 December 2019

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC-BY).

Summary

The need for optimization is almost everywhere today, especially with the rise of deep learning approaches to AI that require searching through millions of neural network weights and at least dozens of architectural hyperparameters. Optimization problems consist of finding a maximum or minimum value of a objective function (in most practical cases, subject to a set of constraints). Depending on the inherent complexity and characteristics of the function to be optimized, there is a diverse armamentarium of ways to attack optimization problems. Methods including Linear and Nonlinear programming, Dynamic programming, Calculus of Variations, and Optimal Control have long been successfully used on smaller, traditional problems and can in some cases be completely dominant in a given area of application.

However, to realize most successful industrial-scale neural networks, various flavors of gradient descent, in conjunction with the backpropagation algorithm and a substantial literature of hard-won tricks for improving training results, have been most meaningfully brought to bear. Evolutionary Computing (EC) is one alternative family of stochastic, biologically-inspired methods that avoids computing gradient information to solve optimization problems, instead imagining solutions as individual genomes competing for life in populations subject to a selection pressure imposed by the objective, or fitness function (A. E. Eiben & Smith, 2003; De Jong, 2006). The advantage of these methods is that the choice of fitness function is unconstrained, enabling ad-hoc sources of fitness information like aesthetic judgments, noisy performance metrics, and undifferentiable processes to guide evolution of all sorts of parametric products: artwork, electrical components, LISP programs, and neural controllers for playing video games. However, the proper role of EC in deep learning (i.e., deep neuroevolution, (Stanley, Clune, Lehman, & Miikkulainen, 2019)) remains a hotly contested issue: for what kinds of problems can evolutionary processes be fast enough in comparison to gradient methods for training model weights directly? Is EC only practically suitable for automatic tuning of high-level hyperparameters, or, failing even that, exploring subjectively-defined trajectories in high-dimensional latent spaces after useful representations (e.g., word embeddings) have already been learned by gradient methods?

Genetic Algorithms (GAs) embody a specific instantiation of EC that tend to be characterized by discrete generations with nearly-complete selection and replacement of the population each generation, with the genomes being vectors of bits, integers, or real values, and the most commonly used selection operation being fitness-proportionate. As in most EC, a GA manages a population of individuals where each individual is represented by their genome and a fitness value. The fitness value is computed at each generation for each of the individuals and the end-goal is to maximize fitness (De Jong, 2006) by selecting high-fitness parents, creating their offspring by sexually reproductive operations such as mutation and crossover, and reconstituting the population. In comparison to other EC techniques, GAs can sometimes more readily regress, drifting away from their maximum realized fitness towards greater sub-optimality. In some cases this is problematic, but in others, particularly for high-dimensional

The Journal of Open Source Software

and complex fitness landscapes, this tolerance for suffering fitness losses facilitates the exploration needed to reach practically satisfactory solutions in the long-term. This tension between fostering solution diversity and the need to protect good (or genotypically or phenotypically novel) solutions leads to the biologically-inspired inbreeding-mitigation strategy of the Island Model (IM). An IM is a collection of islands (or subpopulations), each separately carrying out a GA. The islands provide diversity advantages to the global population by beginning with different initializations and only allowing a controlled amount of migration between populations so that the best solutions do not very rapidly crowd out all competitors. Lineages that are not currently the global best can survive for a while within subpopulations despite a population-wide fitness disadvantage, buying valuable exploration time, and perhaps allowing more fruitful hybrids when mature populations interbreed. The islands choose and transfer some of their individuals to other islands on a specified migration interval, according to a specified migration rule, and sometimes constrained to a specified communications or reproductive-access structure. For example, islands may only be allowed to interact with their close neighbors, based on some incompletely-connected topology such as a ring, a star, or a lattice (Cohoon, Hegde, Martin, & Richards, 1987).

In this work, we introduce a C++ library implementing GA operations and Island Models. We call the library `GAIM`, which simply stands for Genetic Algorithms and Island Models. It supports real-valued/floating-point genomes, rather than the more traditional binary genome, and many of the most widely-used selection, mutation, and crossover operators. More precisely, it implements the following selection operators, (i) roulette-wheel, (ii) rank, (iii) random, (iv) tournament; crossover operators, (i) one-point, (ii) two-points, (iii) uniform, (iv) flat, (v) discrete, and (vi) order-one; and mutation operators, (i) random, (ii) uniform and non-uniform, (iii) fusion, and (iv) swap. The core design principle of `GAIM` is to allow the user to extend this suite of operators with their own custom operators and to allow them to point arbitrarily to an external fitness function (that receives a vector of floating point parameters and returns a single fitness value) of their choice. `GAIM`'s goal is to provide a module to help users quickly integrate fast GAs in their applications, rather than attempting to comprehensively implement the numerous variants of GAs proposed in the literature. Complex frameworks that chronicle these developments and are mostly implemented in interpreted languages tend to impose overhead that is potentially unacceptable when fitness evaluations are very fast and very numerous. With `GAIM`, the user has the ability to execute independent GA runs on different threads, which is very useful for deriving good Monte Carlo estimates of average fitness across independent evolutionary trajectories. Because parallelization of island models is even more inconvenient than a simple GA for an nonspecialist application writer or researcher to add directly to their program, a basic IM implementation is provided. Islands are implemented as threads and concurrency mechanisms are used to synchronize the communication of individual genomes between them only when it is necessary (based on the migration interval). Furthermore, we provide MPI implementations for independent GA runs and for Island Models. Both can be used in case of large-scale optimization problems running on computer clusters or grids.

Finally, because experimentation overwhelmingly revolves around tuning evolution for a fitness function of interest, evolutionary parameters are specified separately (in a file) from the function to be optimized (in users' source code), helping the research-oriented user to keep track of numerous experiments and their results. `GAIM` provides tools for logging and storing information regarding an experiment such as fitness of individuals, the best genome within a population, etc. Furthermore, Python code is provided for plotting and analyzing results.

A small test suite accompanies the project so that `GAIM`'s smooth and robust operation can be easily verified; furthermore, the project benefits from taking advantage of the CI infrastucture provided by GitLab. More details about the implementation and the API available to the user who wishes to integrate `GAIM` into their program are available in documentation. We also furnish short tutorials and examples showing how to make use of `GAIM` either as a standalone executable or as a shared library.

The Journal of Open Source Software

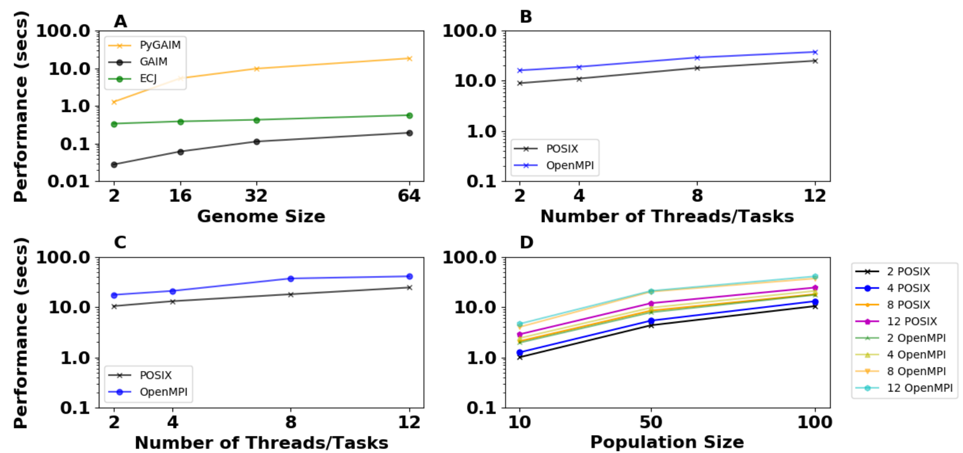

Figure 1: **A** Time for the optimizer to reach the global minimum of the Schwefel function (http://benchmarkfcns.xyz/benchmarkfcns/schwefelfcn.html) is plotted against the dimensionality of the function (size of genome). Separate series show that convergence is faster in terms of real time for GAIM versus a Python implementation and ECJ. Single population of 20 individuals running for 10000 generations for each case of different genome size. **B** Performance of 2, 4, 8, and 12 independent GAs (no communication between threads/processes) running for 10000 generations with 100 individuals of 100 genes each. POSIX threads (black) and MPI processes (blue) lines indicate that POSIX implementation is faster. Both cases scale linear with the number of threads. **C** Performance of Island Model for 2, 4, 8, and 12 islands. The IM runs for 10000 generations with 100 individuals of 100 genes each. **D** Comparison of POSIX threads and MPI processes running for 10000 generations and with varying number of individuals (10, 50, 100). The genome size is 100 in all cases. Each island communicates with all the others since we employ a all-to-all topology (worst case scenario). In all cases the fitness function is the Schwefel test function.

References

A. E. Eiben, & Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer. Retrieved from https://books.google.com/books?id=7IOE5VIpFpwC

Cohoon, J. P., Hegde, S. U., Martin, W. N., & Richards, D. (1987). Punctuated equilibria: A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application* (pp. 148–154). Hillsdale, NJ, USA: L. Erlbaum Associates Inc. Retrieved from http://dl.acm.org/citation.cfm?id=42512.42532

De Jong, K. A. (2006). *Evolutionary computation: A unified approach*. MIT Press. Retrieved from https://books.google.com/books?id=1Yn6AQAAQBAJ

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*(1), 24–35. doi:10.1038/s42256-018-0006-z