operon

HEAL Research

Apr 24, 2024

CONTENTS

Note: The documentation is under construction.

Operon is a modern C++ framework for symbolic regression that uses genetic programming to explore a hypothesis space of possible mathematical expressions in order to find the best-fitting model for a given regression target. Its main purpose is to help develop accurate and interpretable white-box models in areas such as system identification.

CHAPTER

ONE

MOTIVATION

Operon was motivated by the need to have a *flexible* and *performant* system that *works out of the box*. Thus, it was developed with the following goals in mind:

Modern concurrency model
: Traditional threading approaches are not optimal for today's many-core systems. This means designing the evolutionary main loop in such a way as to avoid synchronisation overhead and take advantage of C++17's execution policies.

Performance
: By using an efficient linear tree representation where each `Node` is trivial and vectorized evaluation with the help of the Eigen library. The encoding consumes 40 bytes per tree node, allowing practitioners to work with very large populations.

Ease-of-use
: *Operon* (the core library) comes with a command-line client that just works: you pass it a dataset and it will start optimizing. Its behavior can be configured by command line options, making it easy to integrate with any scripting environment or high-level language such as Python. A Python script is provided for performing experiments automatically aggregating the results.

  For more advanced use cases, we provide a C++ and a Python API, briefly illustrated with some examples.

For an overview of *Operon* please have a look at the *Features* page.

The software was also presented at GECCO'2020 *EvoSoft* workshop: https://dl.acm.org/doi/10.1145/3377929.3398099. If you want to reference it in your publication, please use:

1.1 Reference

```
@inproceedings{Burlacu:2020:GECCOcomp,
author = {Bogdan Burlacu and Gabriel Kronberger and Michael Kommenda},
title = {Operon C++: An Efficient Genetic Programming Framework for Symbolic Regression},
year = {2020},
  editor = {Richard Allmendinger and others},
  isbn13 = {9781450371278},
publisher = {Association for Computing Machinery},
  publisher_address = {New York, NY, USA},
url = {https://doi.org/10.1145/3377929.3398099},
doi = {doi:10.1145/3377929.3398099},
booktitle = {Proceedings of the 2020 Genetic and Evolutionary Computation Conference␣
↪Companion},
pages = {1562-1570},
```

(continues on next page)

(continued from previous page)

```
  size = {9 pages},
  keywords = {genetic algorithms, genetic programming, C++, symbolic regression},
  address = {internet},
series = {GECCO '20},
  month = {July 8-12},
  organisation = {SIGEVO},
  abstract = {},
  notes = {Also known as \cite{10.1145/3377929.3398099}
           GECCO-2020
           A Recombination of the 29th International Conference on Genetic Algorithms␣
↪(ICGA) and the 25th Annual Genetic Programming Conference (GP)},
}
```

1.1.1 Features

Genetic programming is quite computationally intensive, but that doesn't mean it has to be slow. *Operon*'s main goals are to provide excellent performance and to offer a user-friendly experience.

Implementation

- Cross-platform
- Highly-scalable concurrency model using Threading Building Blocks
- Low synchronization overhead via atomic primitives
- Very fast, vectorized model evaluation using Eigen
- Low memory footprint (linear tree encoding as an array of `Node` (40 byte each)
- Support for numerical and automatic differentiation using Ceres
- Nonlinear least squares optimization of model parameters using Ceres
- State of the art numerically stable error metrics (R-Squared, MSE, RMSE, NMSE)
- Python bindings using pybind11
- Modern genetic programming design (see below)

Encoding

- *Operon* uses a linear postfix representation for expression trees
- A `Tree` encapsulates a contiguous array of `Nodes`.
- The `Node` is `trivial` and has `standard_layout`.
- All operators manipulate the representation directly, with no intermediate step.

Logical parallelism

- *Streaming* genetic operators designed to integrate efficiently with the concurrency model
- New individuals are generated concurrently in separate logical threads
- Optimal distribution of logical tasks to physical threads is left to the underlying scheduler
- Consistency and predictability across different machines (to the extent possible)

Evolutionary model

Operon implements an efficient evolutionary model centered around the concept of an *offspring generator* — an operator that encapsulates a preconfigured recipe for producing a single child individidual from the parent population. A general recipe would look like below:

selection crossover mutation evaluation acceptance

Because these operators don't share mutable state, the execution of such pipelines can be easily parallelized, such that each offspring generation event takes place independently in its own *logical thread*.

The *offspring generator* concept allows deploying different evolutionary models (standard, offspring selection, soft brood selection) within the same algorithmic main loop.

Hybridization with local search

Conceptually, this kind of hybridization is useful to shift the effort of finding appropriate numerical coefficients for the model from the evolutionary algorithm itself to a specialized optimization procedure. Algorithms that incorporate some kind of local search optimization step are also called *memetic algorithms*.

Under the hood, we take advantage of Ceres' integration with Eigen in order to optionally perform non-linear least squares fitting of model coefficients during the model evaluation step. The local optimization step is typically quite computationally intensive since model derivatives have to be computed (Ceres supports numerical or automatic differentiation).

In order to promote fair comparison between different algorithmic variants (with and without local search), we allow constraints on an algorithm's evaluation budget – including local search.

Genetic operators

Operators spend from a global evaluation budget, facilitating fair comparisons between algorithmic flavors. All the typical selection and recombination opperations are supported:

Selection

Random, **proportional** and **tournament** selection are suported. In the case of crossover, a separate selection mechanism can be configured for each parent.

Mutation

Mutation can act on a single node or on an entire subtree. Depending on the node type, single-node mutation can change the node value, the data label or the type of mathematical operation performed by the node. Subtree mutation can insert, replace or delete a subtree.

Crossover

The crossover operator generates a single child individual from two parents, by swapping a subtree from the first parent (algo called the *root parent*) with a subtree from the second parent (also called the *non-root parent*). The operator supports a configurable bias towards internal nodes.

Both **crossover** and **mutation** operators are designed to ensure that depth and length restrictions on the tree individuals are always respected.

Tree initialization

Supported algorithms:

- Grow tree creator
- Balanced tree creator (BTC)
- Probabilistic tree creator (PTC)
- Configurable primitive set
- Hybridization with local search
- Novel tree hashing algorithm (paper)
- Fast calculation of population diversity (using tree hashes)

1.1.2 Build instructions

The project requires `CMake` and a `C++17` compliant compiler.

Configuration options

Note: These options are specified to `CMake` in the form `-D<OPTION>=<ON|OFF>`. All options are `OFF` by default. Options that depend on additional libraries require those libraries to be present and detectable `CMake`.

---

1. `USE_SINGLE_PRECISION`: Perform model evaluation using floats (single precision) instead of doubles. Great for reducing runtime, might not be appropriate for all purposes.
2. `USE_OPENLIBM`: Link against Julia's openlibm, a high performance mathematical library (recommended to improve consistency across compilers and operating systems).
3. `BUILD_TESTS`: Build the unit tests.
4. `BUILD_PYBIND`: Build the Python bindings.
5. `USE_JEMALLOC`: Link against jemalloc, a general purpose `malloc(3)` implementation that emphasizes fragmentation avoidance and scalable concurrency support (mutually exclusive with `tcmalloc`).

6. USE_TCMALLOC: Link against tcmalloc (thread-caching malloc), a `malloc(3)` implementation that reduces lock contention for multi-threaded programs (mutually exclusive with *jemalloc*).

Install Examples

Linux/Conda

Here is an example install of operon in linux using a conda environment and bash commands.

1. Specify an *environment.yml* file with the dependencies:

```
name: operon-env
channels:
  - conda-forge
dependencies:
  - python=3.9.1
  - cmake=3.19.1
  - pybind11=2.6.1
  - eigen=3.3.9
  - fmt=7.1.3
  - ceres-solver=2.0.0
  - taskflow=3.1.0
  - openlibm
  - cxxopts
```

2. Run the following commands:

```
# create and activate conda environment
conda env create -f environment.yml
conda activate operon-env

# Use gcc-9 (or later)
export CC=gcc-9
export CXX=gcc-9

# clone operon
git clone https://github.com/heal-research/operon
cd operon

# run cmake with options
mkdir build; cd build;
cmake .. -DCMAKE_BUILD_TYPE=Release  -DBUILD_PYBIND=ON -DUSE_OPENLIBM=ON -DUSE_
↪SINGLE_PRECISION=ON -DCERES_TINY_SOLVER=ON

# build
make VERBOSE=1 -j pyoperon

# install python package
make install
```

3. To test that the python package installed correctly, try `python -c "from operon.sklearn import SymbolicRegressor"`.

Windows/VcPkg

Alternatively, vcpkg also works on Windows and Linux.

1. Install dependencies and clone the repo

```
# install dependencies
vcpkg install ceres:x64-linux fmt:x64-linux pybind11:x64-linux cxxopts:x64-
↪linux doctest:x64-linux python3:x64-linux taskflow:x64-linux

# clone operon
git clone https://github.com/heal-research/operon
cd operon
```

2. Configure and build (make sure to use the appropriate generator for your system, e.g. `-G "Visual Studio 16 2019" -A x64`. If the python path is not correctly detected, you can specify the install destination for the python module with `-DCMAKE_INSTALL_PREFIX=<path>`

```
# configure
mkdir build && cd build
cmake .. -DCMAKE_TOOLCHAIN_FILE=<path-to-vcpkg>/scripts/buildsystems/vcpkg.
↪cmake -DCMAKE_BUILD_TYPE=Release  -DBUILD_PYBIND=ON -DUSE_OPENLIBM=ON -
↪DUSE_SINGLE_PRECISION=ON -DCERES_TINY_SOLVER=ON

# build
make -j pyoperon
```

3. To test that the python package installed correctly, try `python -c "from operon.sklearn import SymbolicRegressor"`.

1.1.3 Example

This example shows how to do symbolic regression using GP. The impatient can go directly to the full code example on github. We assume *some* familiarity with GP concepts and terminology.

We solve a synthetic benchmark problem, namely the *Poly-10*:

$$F(\mathbf{x}) = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$$

This problem consists of 500 datapoints which we'll split equally between our *training* and *test* data.

First, let's load the data from csv into a *Dataset* and set the data partitions:

```
Dataset ds("../data/Poly-10.csv", /* csv has header */ true);
Range trainingRange { 0, ds.Rows() / 2 };
Range testRange     { ds.Rows() / 2, ds.Rows() };
```

By convention, the program will use all the dataset columns (except for the target column) as features. The user is responsible for preprocessing the data prior to the modeling step.

Next, we define the optimization target and create a `Problem`:

```
const std::string target = "Y";
Problem problem(ds, ds.Variables(), target, trainingRange, testRange);
problem.GetGrammar().SetConfig(Grammar::Arithmetic);
```

In the snippet above, `problem.GetGrammar().SetConfig(Grammar::Arithmetic)` configures the problem to use an arithmetic grammar, consisting of the symbols $+, -, \times, \div$. The `Grammar` class keeps track of the allowed symbols and their initial frequencies (taken into account when the population is initialized).

In what follows, we define the *genetic operators*. Although the code is a bit verbose, it's purpose should be clear.

Crossover and mutation

The so-called recombination operators generate offspring individuals by combining genes from the parents (crossover) and adding random perturbations (mutation).

- The *crossover* operator takes the tree *depth* and *length* limits and generates offspring that do not exceed them. It can be further parameterized by an *internal node bias* parameter which controls the probability of selecting an internal node (eg., a function node) as a cut point.
- The *mutation* operator can apply different kind of perturbations to a tree individual: *point mutation* changes a leaf node's coefficient (eg., a variable node's *weight* or a constant node's *value*), *change variable* mutation changes a variable with another one from the dataset (eg., $x_1 \rightarrow x_2$) and the *change function* mutation does the same thing to function symbols.

```
// set up crossover and mutation
double internalNodeBias = 0.9;
size_t maxTreeDepth  = 10;
size_t maxTreeLength = 50;
SubtreeCrossover crossover { internalNodeBias, maxTreeDepth, maxTreeLength };
MultiMutation mutation;
OnePointMutation onePoint;
ChangeVariableMutation changeVar { problem.InputVariables() };
ChangeFunctionMutation changeFunc { problem.GetGrammar() };
mutation.Add(onePoint, 1.0);
mutation.Add(changeVar, 1.0);
mutation.Add(changeFunc, 1.0);
```

Selector

The selection operator samples the distribution of fitness values in the population and picks parent individuals for taking part in recombination. *Operon* supports specifying different selection methods for the two parents (typically called *male* and *female* or *root* and *non-root* parents). We tell the selector how to compare individuals by providing a lambda function to its constructor:

```
// our lambda function simply compares the fitness of the individuals
auto comp = [](Individual const& lhs, Individual const& rhs) {
    return lhs[0] < rhs[0];
};
// set up the selector
TournamentSelector selector(comp);
selector.TournamentSize(5);
```

Evaluator

This operator is responsible for calculating fitness and is alloted a fixed evaluation budget at the beginning of the run. The *evaluator* is also capable of performing nonlinear least-squares fitting of model parameters if the *local optimization iterations* parameter is set to a value greater than zero.

```
// set up the evaluator
RSquaredEvaluator evaluator(problem);
evaluator.LocalOptimizationIterations(config.Iterations);
evaluator.Budget(config.Evaluations);
```

Reinserter

The reinsertion operator merges the pool of *recombinants* (new offspring) back into the population. This can be a simple replacement or a more sophisticated strategy (eg., keep the best individuals among the parents and offspring). Like the selector, the reinserter requires a lambda to specify how it should compare individuals.

```
ReplaceWorstReinserter<> reinserter(comp);
```

Offspring generator

Implements a strategy for producing new offspring. This can be plain recombination (eg., crossover + mutation) or more elaborate logic like acceptance criteria for offspring or brood selection. In general, this operation may *fail* (returning a *maybe* type) and should be handled by the algorithm designer.

```
// the generator makes use of the other operators to generate offspring and assign␣
↪fitness
// the selector is passed twice, once for the male parent, once for the female parent.
BasicOffspringGenerator generator(evaluator, crossover, mutation, selector, selector);
```

Tree creator

The tree creator initializes random trees of any target length. The length is sampled from a uniform distribution $U[1, maxTreeLength]$. Maximum depth is fixed by the *maxTreeDepth* parameter.

```
// set up the solution creator
std::uniform_int_distribution<size_t> treeSizeDistribution(1, maxTreeLength);
BalancedTreeCreator creator { treeSizeDistribution, maxTreeDepth, maxTreeLength };
```

Finally, we can configure the genetic algorithm and run it. A callback function can be provided to the algorithm in order to report progress at the end of each generation.

```
GeneticAlgorithmConfig config;
config.Generations          = 100;
config.PopulationSize       = 1000;
config.PoolSize             = 1000;
config.Evaluations          = 1000000;
config.Iterations           = 0;
config.CrossoverProbability = 1.0;
config.MutationProbability  = 0.25;
config.Seed                 = 42;

// set up a genetic programming algorithm
GeneticProgrammingAlgorithm gp(problem, config, creator, generator, reinserter);

int generation = 0;
auto report = [&] { fmt::print("{}\n", ++generation); };
Random random(config.Seed);
gp.Run(random, report);
```