Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations

Oskar Taubert(✉), Marie Weiel, Daniel Coquelin, Anis Farshian, Charlotte Debus, Alexander Schug, Achim Streit, and Markus Götz

Steinbuch Centre for Computing (SCC), Karlsruhe Institute of Technology (KIT), 76344 Eggenstein-Leopoldshafen, Germany
{oskar.taubert,markus.goetz}@kit.edu

Abstract. We present Propulate, an evolutionary optimization algorithm and software package for global optimization and in particular hyperparameter search. For efficient use of HPC resources, Propulate omits the synchronization after each generation as done in conventional genetic algorithms. Instead, it steers the search with the complete population present at time of breeding new individuals. We provide an MPI-based implementation of our algorithm, which features variants of selection, mutation, crossover, and migration and is easy to extend with custom functionality. We compare Propulate to the established optimization tool Optuna. We find that Propulate is up to three orders of magnitude faster without sacrificing solution accuracy, demonstrating the efficiency and efficacy of our lazy synchronization approach. Code and documentation are available at https://github.com/Helmholtz-AI-Energy/propulate/.

Keywords: Genetic Optimization · AI · Parallelization · Evolutionary Algorithm

1 Introduction

Machine learning (ML) algorithms are heavily used in almost every area of human life today, from medical diagnosis and critical infrastructure to transportation and food production. Almost all ML algorithms have non-learnable hyperparameters (HPs) that influence the training and in particular their predictive capacity. As evaluating a set of HPs involves at least a partial training, state-free approaches to HP optimization (HPO), like grid and random search, often go beyond available compute resources [15]. To explore the high-dimensional HP spaces efficiently, information from previous evaluations must be leveraged to guide the search. Such state-dependent strategies minimize the number of evaluations to find a useful model, reducing search times and thus the energy consumption of the computation. Bayesian and bio-inspired optimizers are the most popular of these AutoML approaches. Among the latter, genetic

A. Bhatele et al. (Eds.): ISC High Performance 2023, LNCS 13948, pp. 106–124, 2023.
https://doi.org/10.1007/978-3-031-32041-5_6

algorithms (GAs) are versatile metaheuristics inspired by natural evolution. To solve a search-for-solutions problem, a population of candidate solutions (or individuals) is evolved in an iterative interplay of selection and variation [23,30]. Although reaching the global optimum is not guaranteed, GAs often find near-optimal solutions with less computational effort than classical optimizers [8,9]. They have become popular for various optimization problems, including HPO for ML and neural architecture search (NAS) [14].

To take full advantage of the increasingly bigger models and datasets, designing scalable algorithms for high performance computing (HPC) has become a must [40]. While Bayesian optimization is inherently serial, the structure of GAs renders them suitable for parallelization [34]: Since all candidates in each iteration are independent, they can be evaluated in parallel. To breed the next generation, however, the previous one has to be completed. As the computational expenses for evaluating different candidates vary, synchronizing the parallel evolutionary process affects the scalability by introducing a substantial bottleneck. Approaches to reducing the overall communication in parallel GAs like the island model (IM) [34] do not address the underlying synchronization problem.

To solve the issues arising from explicit synchronization, we introduce Propulate, a massively parallel genetic optimizer with asynchronous propagation of populations and migration. Unlike classical GAs, Propulate maintains a continuous population of already evaluated individuals with a softened notion of the typically strictly separated, discrete generations. Our contributions include:

- A novel parallel genetic algorithm based on a fully asynchronous island model with independently processing workers, allowing to parallelize the optimization process and distribute the internal evaluation of the objective function.
- Massive parallelism by asynchronous propagation of continuous populations and migration.
- A prototypical implementation in Python using extremely efficient communication via the message passing interface (MPI).
- Optimal use of parallel hardware by minimizing idle times in HPC systems.

We use Propulate to optimize various benchmark functions and the HPs of a deep neural network on a supercomputer. Comparing our results to those of the popular HPO package `Optuna`, we find that Propulate is consistently drastically faster without sacrificing solution accuracy. We further show that Propulate scales well to at least 100 processing elements (PEs) without relevant loss of efficiency, demonstrating the efficacy of our asynchronous evolutionary approach.

2 Related Work

Recent progress in ML has triggered heavy use of these techniques with Python as the de facto standard programming language. Tuning HPs requires solving high-dimensional optimization problems with ML algorithms as black boxes and model performance metrics as objective functions (OFs). Most common are Bayesian optimizers (e.g. `Optuna` [2], `Hyperopt` [7], `SMAC3` [24,27], `Spearmint` [32], `GPyOpt` [5], and `MOE` [38]) and bio-inspired methods such as

swarm-based (e.g. FLAPS [39]) and evolutionary (e.g. DEAP [16], MENNDL [40]) algorithms. Below, we provide an overview of popular HP optimizers in Python, with a focus on state-dependent parallel algorithms and implementations. A theoretical overview of parallel GAs can be found in surveys [3,4,12] and books [29,37].

Optuna adopts various algorithms for HP sampling and pruning of unpromising trials, including tree-structured Parzen estimators (TPEs), Gaussian processes, and covariance matrix adaption evolution strategy. It enables parallel runs via a relational database server. In the parallel case, an Optuna candidate obtains information about previous candidates from and stores results to disk.

SMAC3 (**S**equential **M**odel-based **A**lgorithm **C**onfiguration) combines a random-forest based Bayesian approach with an aggressive racing mechanism [24]. Its parallel variant pSMAC uses multiple collaborating SMAC3 runs which share their evaluations through the file system.

Spearmint, GPyOpt, and MOE are Gaussian-process based Bayesian optimizers. Spearmint enables distributed HPO via Sun Grid Engine and MongoDB. GPyOpt is integrated into the Sherpa package [22], which provides implementations of recent HP optimizers along with the infrastructure to run them in parallel via a grid engine and a database server. MOE (**M**etric **O**ptimization **E**ngine) uses a one-step Bayes-optimal algorithm to maximize the multi-points expected improvement in a parallel setting [38]. Using a REST-based client-server model, it enables multi-level parallelism by distributing each evaluation and running multiple evaluations at a time.

Nevergrad [31] and Autotune [25] provide gradient-free and evolutionary optimizers, including Bayesian, particle swarm, and one-shot optimization. In Nevergrad, parallel evaluations use several workers via an executor from Python's concurrent module. Autotune enables concurrent global and local searches, cross-method sharing of evaluations, method hybridization, and multi-level parallelism. Open Source Vizier [33] is a Python interface for Google's HPO service Vizier. It implements Gaussian process bandits [19] and enables dynamic optimizer switching. A central database server does the algorithmic proposal work, clients perform evaluations and communicate with the server via remote procedure calls. Katib [18] is a cloud-native AutoML project based on the Kubernetes container orchestration system. It integrates with Optuna and Hyperopt. Tune [26] is built on the Ray distributed computing platform. It interfaces with Optuna, Hyperopt, and Nevergrad and leverages multi-level parallelism.

DEAP (**D**istributed **E**volutionary **A**lgorithms in **P**ython) [16] implements general GAs, evolution strategies, multi-objective optimization, and co-evolution of multi-populations. It enables parallelization via Python's multiprocessing or SCOOP module. EvoTorch [36] is built on PyTorch and implements distribution- and population-based algorithms. Using a Ray cluster, it can scale over multiple CPUs, GPUs, and computers. MENNDL (**M**ulti-node **E**volutionary **N**eural **N**etworks for **D**eep **L**earning) [40] is a closed-source MPI-parallelized HP optimizer for automated network selection. A master node handles the genetic operations while evaluations are done on the remaining worker nodes. However, global synchronization hinders optimal resource utilization [40].

Algorithm 1: Basic GA. In each generation, the individuals are evaluated in terms of the optimization problem's OF. Genetic operators propagate them to the next generation: The selection operator chooses a portion of the current generation, where better individuals are usually preferred. To breed new individuals, the genes of two or more parent individuals from the selected pool are manipulated. While the crossover operator recombines the parents' genes, the mutation operator alters them randomly. This is repeated until a stopping condition is met.

Input: Search-space limits, population size P, *termination_condition*, *selection_policy*, *crossover_probability*, *mutation_probability*.

Initialize population *pop* of P individuals within search space.
while *not* *termination_condition* **do** `// OPTIMIZE`
  Evaluate individuals in *pop*. `// EVALUATE`
  Choose *parents* from *pop* following *selection_policy*. `// SELECT`
  foreach *individual* ***in*** *pop* **do** `// VARY`
    if *random* $\leq$ *crossover_probability* **then** `// RECOMBINE`
      Recombine individuals randomly chosen from *parents*.
    if *random* $\leq$ *mutation_probability* **then** `// MUTATE`
      Mutate.
    Update *individual* in *pop*.

Result: Best individual found (i.e., with lowest OF value for minimization).

3 Propulate Algorithm and Implementation

To alleviate the bottleneck inherent to synchronized parallel genetic algorithms, our massively parallel genetic optimizer Propulate (***prop****agate* and *pop****ulate***) implements a fully asynchronous island model specifically designed for large-scale HPC systems. Unlike conventional GAs, Propulate maintains a continuous population of evaluated individuals with a softened notion of the typically strictly separated generations. This enables *asynchronous* evaluation, variation, propagation, and migration of individuals. To ensure interoperability with existing data science and ML workflows, we provide a Python implementation. In most applications, evaluating the OF represents the largest contribution to the total resource consumption. Performance-relevant paths inside the OF evaluation are expected to be implemented and optimized in CUDA and C/C++ or Fortran. With the aforementioned workflows, this is typically already the case.

Propulate's basic mechanism is that of Darwinian evolution, i.e., beneficial traits are selected, recombined, and mutated to breed more fit individuals (see Algorithm 1). On a higher level, Propulate employs an IM, which combines independent evolution of self-contained subpopulations with intermittent exchange of selected individuals [34]. To coordinate the search globally, each island occasionally delegates migrants to be included in the target islands' populations. With worse performing islands typically receiving candidates from better performing ones, islands communicate genetic information competitively, thus increasing diversity among the subpopulations compared to panmictic models [11]. Independent from the breeding mechanism used on each single island of a synchronous

IM, this migrant exchange occurs simultaneously after a fixed number of synchronously evaluated generations, with no computation happening in that time. The following hyperparameters characterize an IM:

- **Island number and subpopulation sizes**
- **Migration (pollination) probability**
- **Number of migrants (pollinators):** How many individuals migrate from the source population at a time.
- **Migration (pollination) topology:** Directed graph of migration (pollination) paths between islands.
- **Emigration policy:** How to select emigrants (e.g., random or best) and whether to remove them from the source population (actual migration) or not (pollination).
- **Immigration policy:** How to insert immigrants into the target population, i.e., either add them (migration) or replace existing individuals (pollination, e.g., random or worst).

Propulate's functional principle is outlined in Algorithm 2. We consider multiple PEs (or workers) partitioned into islands. Each worker processes one individual at a time and maintains a population to track evaluated and migrated individuals on its island. To mitigate the computational overhead of synchronized OF evaluations, Propulate leverages asynchronous propagation of continuous populations with interwoven, worker-specific generations (see Fig. 1). In each iteration, each worker breeds and evaluates an individual which is added to its population list. It then sends the individual with its evaluation result to all workers on the same island and, in return, receives evaluated individuals dispatched by them for a mutual update of their population lists. To avoid explicit synchronization points, the independently operating workers use asynchronous point-to-point communication via MPI to share their results. Each one dispatches its result immediately after finishing an evaluation. Directly afterwards, it non-blockingly checks for incoming messages from workers of its own island awaiting to be received. In the next iteration, it breeds a new individual by applying the evolutionary operators to its continuous population list of all evaluated individuals from any generation on the island. The workers thus proceed asynchronously without idle times despite the individuals' varying computational costs.

After the mutual update, asynchronous migration or pollination between islands happens on a per-worker basis with a certain probability. Each worker selects a number of emigrants from its current population. For actual migration[1], an individual can only exist actively on one island. A worker thus may only choose eligible emigrants from an exclusive subset of the island's population to avoid overlapping selections by other workers. It then dispatches the emigrants to the target islands' workers as specified in the migration topology. Finally, it sends them to all workers on its island for island-wide deactivation of emigrated individuals before deactivating them in its own population.

[1] See github.com/Helmholtz-AI-Energy/propulate/tree/master/supplementary for pseudocode with migration and explanatory figure.

Algorithm 2: Propulate with pollination.

Input: Search-space limits; hyperparameters *n_islands*, island sizes P_i ($i = 1, \ldots,$ *n_islands*), number of iterations *generations*, evolutionary operators (including *selection_policy*, *crossover_probability*, *mutation_probability* etc.), *pollination_probability*, *pollination_topology*, *emigration_policy*, *immigration_policy*.

Configure *n_islands* islands with P_i workers each. Each worker evaluates one individual at a time and maintains its own population list *pop* of evaluated and migrated individuals on the island.
`/* START OPTIMIZATION. */`
for each worker **do in parallel**
  while *generation* $\leq$ *generations* **do** `// Loop over generations.`
    Breed and evaluate individual. Append it to *pop*. Send it to other workers on island to synchronize their populations lists: `evaluate_individual()` `// BREED AND EVALUATE`
    Check for and possibly receive individuals bred and evaluated by other workers on island. Append them to *pop*: `receive_intra_isle_individuals()` `// SYNCHRONIZE`
    if *random* $\leq$ *pollination_probability* **then** `// EMIGRATE`
      Choose pollinators from currently active individuals on island according to *emigration_policy*. Send copies of pollinator(s) to workers of target islands according to *pollination_topology*: `send_emigrants()`
    Check for and possibly receive pollinators sent by workers from other islands. Add them to *pop*. Determine individuals to be replaced by incoming pollinators according to *immigration_policy*. Send individuals to be replaced to other workers on island for deactivation: `receive_immigrants()` `// IMMIGRATE`
    Check for and possibly receive individuals replaced by pollinators on other workers on island. Try to deactivate them in *pop*. If an individual to be deactivated is not yet in *pop*, append it to history list *replaced* and try again in the next generation: `deactivate_replaced_individuals()` `// SYNCHRONIZE`
    Go to next generation: `generation += 1`
  `/* OPTIMIZATION DONE: FINAL SYNCHRONIZATION */`
  Wait for all other workers to finish: `MPI.COMM_WORLD.barrier()`
  Final check for incoming messages so all workers hold complete population.
  Probe for individuals evaluated by other workers on island: `receive_intra_isle_individuals()`
  Probe for incoming pollinators immigrating from other islands: `receive_immigrants()`
  Probe for individuals replaced by other workers on island to be deactivated: `deactivate_replaced_individuals()`

Result: n individuals with smallest OF values.

In the next step, the worker probes for and, if applicable, receives immigrants from other islands. It then checks for individuals emigrated by other workers of its island and tries to deactivate them in its population. Due to the asynchronicity,

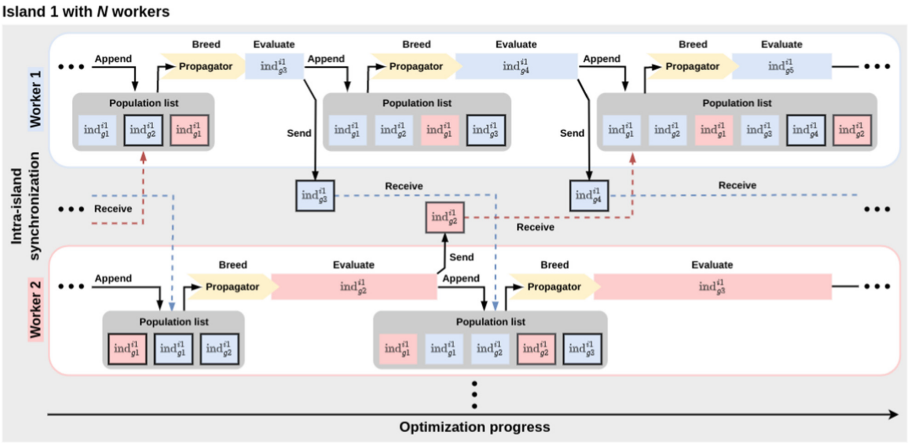

Fig. 1. Asynchronous propagation. Interaction of two workers on one island. Individuals bred by worker 1 and 2 are shown in blue and red, respectively. Their origins are given by a generation sub- and an island superscript. Populations are depicted as round grey boxes, where most recent individuals have black outlines. Varying evaluation times are represented by sharp boxes of different widths. We illustrate the asynchronous propagation and intra-island synchronization of the population using the example of the blue individual ind_{g3}^{i1}. This individual is bred by worker 1 in generation 3 by applying the propagator (yellow) to the worker's current population. After evaluating ind_{g3}^{i1}, worker 1 sends it to all workers on its island and appends it to its population. As no evaluated individuals dispatched by worker 2 await to be received, worker 1 proceeds with breeding. Worker 2 receives the blue ind_{g3}^{i1} only after finishing the evaluation of the red ind_{g2}^{i1}. It then appends both to its population and breeds a new individual for generation 3. (Color figure online)

individuals might be designated to be deactivated before arriving in the population. Propulate continuously corrects these synchronization artefacts during the optimization.

For pollination (see Fig. 2), identical copies of individuals can exist on multiple islands. Workers thus can choose emigrating pollinators from any active individuals in their current populations and do not deactivate them upon emigration. To control the population growth, pollinators replace active individuals in the target population according to the immigration policy. For proper accounting of the population, one random worker of the target island selects the individual to be replaced and informs the other workers accordingly. Individuals to be deactivated that are not yet in the population are cached to be replaced in the next iteration. This process is repeated until each worker has evaluated a set number of generations. Finally, the population is synchronized among workers and the best individuals are returned.

Propulate uses so-called propagators to breed child individuals from an existing collection of parent individuals. It implements various standard genetic operators, including uniform, best, and worst selection, random initialization,

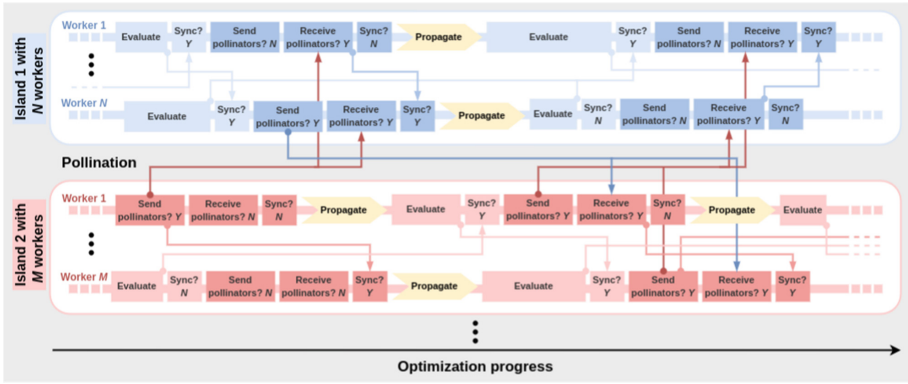

Fig. 2. Asynchronous pollination. Consider two islands with N (blue) and M (red) workers, respectively. We illustrate pollination (dark colors) by tracing worker N on island 1. After evaluation and mutual intra-island updates (light blue, see Fig. 1), this worker performs pollination: It sends copies of the chosen pollinators to all workers of each target island, here island 2. The target island's workers receive the pollinators asynchronously (dark blue arrows). For proper accounting of the populations, worker 1 on island 2 selects the individual to be replaced and informs all workers on its island accordingly (middle red arrow). Afterwards, worker N receives incoming pollinators from island 2 to be included into its population. It then probes for individuals that have been replaced by other workers on its island, here worker 1, in the meantime and need to be deactivated. After these pollination-related intra-island population updates, it breeds the next generation. As pollination does not occur in this generation, it directly receives pollinators from island 2. This time, worker N chooses the individual to be replaced. (Color figure online)

stochastic and conditional propagators, point and interval mutation, and several forms of crossover. In addition, Propulate provides a default propagator: Having selected two random parents from the breeding pool consisting of a set number of the currently most fit individuals, uniform crossover and point mutation are performed each with a specified probability. Afterwards, interval mutation is performed. To prevent premature trapping in a local optimum, a randomly initialized individual is added with a specified probability instead of one bred from the current population.

4 Experimental Evaluation

We evaluate Propulate on various benchmark functions (see Sect. 4.4) and an HPO use case in remote sensing classification (see Sect. 4.5) which provides a real world application. We compare our results against `Optuna` since it is the most widely used HPO software.

4.1 Experimental Environment

We ran the experiments on the distributed-memory, parallel hybrid supercomputer *Hochleistungsrechner Karlsruhe* (HoreKa[2]) at the Steinbuch Centre for Computing, Karlsruhe Institute of Technology. Each of its 769 compute nodes is equipped with two 38-core Intel Xeon Platinum 8368 processors at 2.4 GHz base and 3.4 GHz maximum turbo frequency, 256 GB (standard) or 512 GB (high-memory and accelerator) local memory, a local 960 GB NVMe SSD disk, and two network adapters. 167 of the nodes are accelerator nodes each equipped with four NVIDIA A100-40 GPUs with 40 GB memory connected via NVLink. Inter-node communication uses a low-latency, non-blocking NVIDIA Mellanox InfiniBand 4X HDR interconnect with 200 Gbit/s per port. A Lenovo Xclarity controller measures full node energy consumption, excluding file systems, networking, and cooling. The operating system is Red Hat Enterprise Linux 8.2.

4.2 Benchmark Functions

Benchmark functions are used to evaluate optimizers in terms of convergence, accuracy, and robustness. The informative value of such studies is limited by how well we understand the characteristics making real-life optimization problems difficult and our ability to embed these features into benchmark functions [28]. We use Propulate to optimize a variety of traditional and recent benchmark functions emulating situations optimizers have to cope with in different kinds of problems (see Table 1).

- **Sphere** is smooth, unimodal, strongly convex, symmetric, and thus simple.
- **Rosenbrock** has a narrow minimum inside a parabola-shaped valley.
- **Step** represents the problem of flat surfaces. Plateaus pose obstacles to optimizers as they lack information about which direction is favorable.
- **Quartic** is a unimodal function padded with Gaussian noise. As it never returns the same value on the same point, algorithms that do not perform well on this test function will do poorly on noisy data.
- **Rastrigin** is non-linear and highly multimodal. Its surface is determined by two external variables, controlling the modulation's amplitude and frequency. The local minima are located at a rectangular grid with size 1. Their functional values increase with the distance to the global minimum.
- **Griewank**'s product creates sub-populations strongly codependent to parallel GAs, while the summation produces a parabola. Its local optima lie above parabola level but decrease with increasing dimensions, i.e., the larger the search range, the flatter the function.
- **Schwefel** has a second-best minimum far away from the global optimum.
- **Lunacek's bi-sphere**'s [28] landscape structure is the minimum of two quadratic functions, each creating a single funnel in the search space. The spheres are placed along the positive search-space diagonal, with the optimal and sub-optimal sphere in the middle of the positive and negative quadrant,

[2] https://www.scc.kit.edu/en/services/horeka.php.

Table 1. Benchmark functions.

Name	Function	Limits	Global minimum
Sphere	$f_1 = x_1^2 + x_2^2$	± 5.12	$f(0,0) = 0$
Rosenbrock	$f_2 = 100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$	± 2.048	$f(1,1) = 0$
Step	$f_3 = \sum_{i=1}^{5} \text{int}\,(x_i)$	± 5.12	$f(x_i \leq -5) = -25$
Quartic	$f_4 = \sum_{i=1}^{30}\left(i x_i^4 + \mathcal{N}_i\,(0,1)\right)$	± 1.28	$f(0,...,0) = \sum_i \mathcal{N}_i$
Rastrigin	$f_5 = 200 + \sum_{i=1}^{20} x_i^2 - 10\cos(2\pi x_i)$	± 5.12	$f(0,...,0) = 0$
Griewank	$f_6 = 1 + \frac{1}{4000}\sum_{i=1}^{10} x_i^2 - \prod_{i=1}^{10}\cos\frac{x_i}{\sqrt{i}}$	± 600	$f(0,...,0) = 0$
Schwefel	$f_7 = 10V - \sum_{i=1}^{10} x_i \sin\sqrt{\lvert x_i\rvert}$ with $V = 418.982887$	± 500	$f\left(x_1^*,...,x_{10}^*\right) = 0$, $x_i^* = 420.968746$
Bi-sphere	$f_8 = \min\left(\sum_{i=1}^{30}(x_i - \mu_1)^2,\right.$ $\left.30 + s\cdot\sum_{i=1}^{30}(x_i - \mu_2)^2\right)$ with $\mu_1 = 2.5$, $\mu_2 = -\left(s^{-1}\left(\mu_1^2 - 1\right)\right)^{1/2}$, $s = 1 - \left(2\sqrt{50} - 8.2\right)^{-1/2}$	± 5.12	$f(\mu_1,...,\mu_1) = 0$
Bi-Rastrigin	$f_9 = f_8 + 10\sum_{i=1}^{30} 1 - \cos 2\pi\,(x_i - \mu_1)$	± 5.12	$f(\mu_1,...,\mu_1) = 0$

respectively. Their distance and the barrier's height increase with dimensionality, creating a globally non-separable underlying surface.

– **Lunacek's bi-Rastrigin** [28] is a double-funnel version of Rastrigin. This function isolates global structure as the main difference impacting problem difficulty on a well understood test case.

4.3 Meta-optimizing the Optimizer

Propulate itself has HPs influencing its optimization behavior, accuracy, and robustness. To explore their effect systematically and give transparent recommendations for default values, we conducted a grid search across the six most prominent HPs. The search space is shown in Table 2. We ran the grid search five times for the quartic, Rastrigin, and bi-Rastrigin benchmark functions (see Table 1 and Sect. 4.4), each with a different seed consistently used over all points within a search. All three functions have their global minimum at zero. They were chosen for their high-dimensional parameter spaces (30, 20, and 30, respectively) and different levels of difficulty to optimize. For quartic, Propulate found a minimum below 0.01 ± 0.005 for 80.12% of all points across the five grid searches. This increases to 94.94% for minima found within 0.1 ± 0.05 of the global minimum. In comparison, the tolerances have to be relaxed considerably for the more complex Rastrigin and bi-Rastrigin. While only 18.57% of all grid points had a function value less than 1.0 ± 0.5 for Rastrigin, only a single point resulted in an average value of less than 10 for bi-Rastrigin. Although the average value of bi-Rastrigin was only less than 10 once, we found the minimum across each of the five searches to be less than 1.0 for 3.31% of the grid points.

Table 2. Grid search parameters. All experiments use 144 CPUs equally distributed between two nodes. Random-initialization probability refers to the chance that a new individual is generated entirely randomly.

Number of islands	2	4	8	16	32
Island population size	72	36	18	9	4
Migration (pollination) probability	0.1	0.3	0.5	0.7	0.9
Pollination	True		False		
Crossover probability	0.1	0.325	0.55	0.775	
Point-mutation probability	0.1	0.325	0.55	0.775	
Random-initialization probability	0.1	0.325	0.55	0.775	

Considering grid points with at least one result smaller than 1.0, 86.61% used either 16 or 36 islands, while the remainder used eight. As Propulate initializes different islands at different positions in the search space, the chance that one of them is at a very beneficial position increases with the number of islands. This is further confirmed by a migration probability of 0.7 or 0.9 for 61.41% of these points. If one of the islands is well-initialized, it thus will quickly notify others.

With every best grid point using pollination, we clearly find pollination to be favorable over real migration. To determine the other HPs, we compute the averages of the results for the top ten grid points across all three functions. The top ten were determined by grouping over the lowest average and standard deviation of the function values, sorting by the averages, and sorting by the standard deviations. This method reduces the chances of a single run simply benefiting from an advantageous starting seed. Average crossover, point-mutation, and random-initialization probabilities are 0.655 ± 0.056, 0.363 ± 0.133, and 0.423 ± 0.135, respectively. The average number of islands was 28.800 ± 6.009 which equates to an island population of 5.00 ± 1.043. The average migration probability was 0.527 ± 0.150. These values provide a reasonable starting point towards choosing default HPs for Propulate (see Table 3). As the grid searches only considered functions with independent parameters, we assume a relatively high random-initialization probability to be useful due to the benefits of random search [6]. On this account, we chose to reduce the default random-initialization probability to 0.2. As the migration probability might also be lowered artificially by this phenomenon, we set its default to 0.7. The default probabilities for crossover and point-mutation were chosen as 0.7 and 0.4, respectively. The island size was set at four individuals. This is a practical choice as our test system has four accelerators per node and the number of CPUs per node is a multiple of four.

Table 3. Propulate HPs for benchmark function minimization.

Number of islands	38
Island population size	4
Pollination probability	0.7
Crossover probability	0.7
Point-mutation probability	0.4
Sigma factor	0.05
Random-initialization probability	0.2
Generations per worker	256
Selection policy	Best
Pollination topology	Fully connected
Number of migrants	1
Emigration policy	Best
Immigration policy	Worst

4.4 Benchmark Function Optimization

For each function, we ran each ten equivalent Propulate and `Optuna` optimizations, using the same compute resources, degree of parallelization, and number of evaluations. Figure 3 shows the optimization accuracy over wallclock time comparing Propulate with default parameters determined from our grid search (see Table 3) to `Optuna`'s default optimizer. In terms of accuracy, Propulate and `Optuna` are comparable in most experiments. For many functions, e.g. Schwefel, bi-Rastrigin, and Rastrigin, Propulate even achieves a better OF value. In terms of wallclock time, Propulate is consistently at least one order of magnitude faster. This is due to Propulate's MPI-based communication over the fast network, whereas `Optuna` uses relational databases with SQL and is limited by the slow file system. Since the functions are cheap to evaluate, optimization and communication dominate the wallclock time. In particular for problems where evaluations are cheap compared to the search itself, we find that `Optuna`'s computational efficiency suffers massively from the frequent file locking inherent to its parallelization strategy, reducing its usability for large-scale HPC applications.

In addition, we inspected the evolution of the population over wallclock time for both Propulate and `Optuna`. An example for minimizing the Rastrigin function is shown in Fig. 4. Propulate is roughly three orders of magnitude faster and makes significantly greater progress in terms of both OF values and distance to the global optimum. Due to this drastic difference in runtime, we measured only 46.27 Wh for Propulate compared to `Optuna`'s 2646.29 Wh.

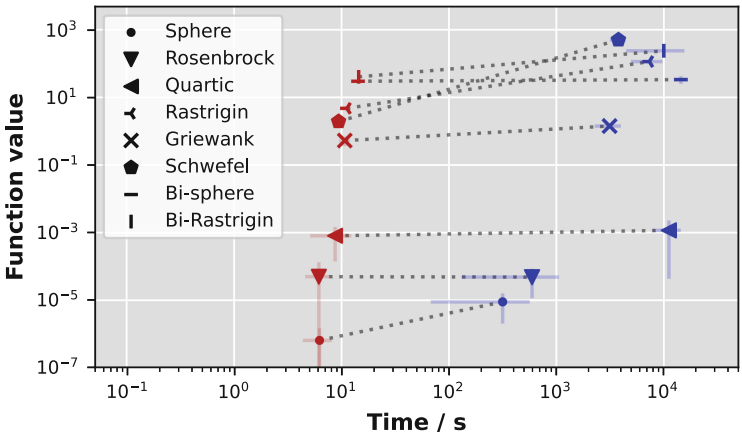

Fig. 3. Benchmark function minimization accuracy over wallclock time. Lowest function values found by Propulate (red) and `Optuna` (blue) versus wallclock time to reach them, each averaged over ten runs. Step is not shown since both optimizers achieve a perfect value of −25 within 0.6 s and 278.2 s, respectively. (Color figure online)

4.5 HP Optimization for Remote Sensing Classification

BigEarthNet [35] is a Sentinel-2 multispectral image dataset in remote sensing. It comprises 590 326 image patches each of which is assigned one or more of the 19 available CORINE Land Cover map labels [10,35]. Multiple computer vision networks for BigEarthNet classification have been trained [35], with ResNet-50 [20] being the most accurate. While a previous Propulate version was used to optimize a set of HPs and the architecture for this use case [13], a more versatile and efficient parallelization strategy in the current version makes it worthwhile to revisit this application. Analogously to [13], we consider different optimizers, learning rate (LR) schedulers, activation functions, loss functions, number of filters in each convolutional block, and activation orders [21]. The search space is shown in Table 4. Optimizer parameters, LR functions, and LR warmup are included as well. We only consider SGD-based optimizers as they share common parameters and thus exclude Adam-like optimizers from the search. We theorize that including Adam led to the difficulties seen previously [13]. The training is exited if the validation loss has not been increasing for ten epochs. We prepared the data analogously to [13]. The network is implemented in TensorFlow [1].

For both Propulate and `Optuna`, we ran each three searches over 24 h on 32 GPUs. We use $1 - F_1^{\text{val}}$ with the validation F_1 score as the OF to be minimized. On average, `Optuna` achieves its best OF value of (0.39 ± 0.01) within (7.05 ± 3.14) h. Propulate beats `Optuna`'s average best after (5.30 ± 2.41) h and achieves its best OF value of (0.36 ± 0.00) within (13.89 ± 5.15) h.

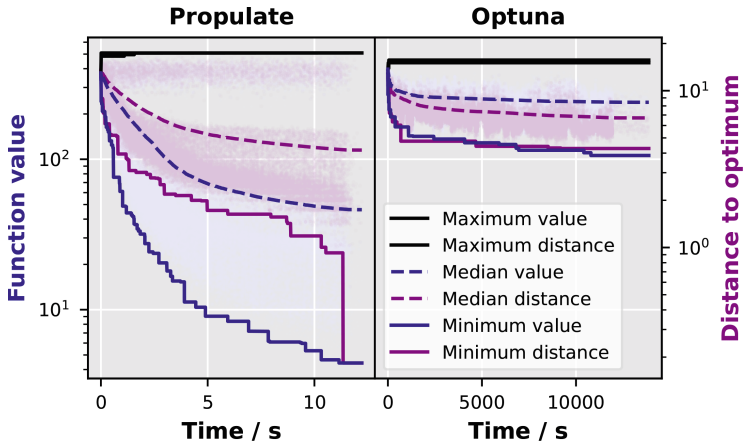

Fig. 4. Evolution of the population over wallclock time for the Rastrigin function. Propulate (left) versus `Optuna` (right). OF values (blue) use the left-hand scale, distances to the global optimum (purple) use the right-hand scale. Pastel dots show each individual's OF value/distance. Solid (dashed) lines show the minimum (median) value and distance achieved so far. Maximum value and distance are shown in black. Both optimizers perform 38 912 evaluations. Note the difference on the time axis. (Color figure online)

4.6 Scaling

Finally, we explore Propulate's scaling behavior for the use case presented in Sect. 4.5. Figure 5 shows our results for weak and strong linear scaling. Our baseline configuration used two nodes. Since each node has four GPUs, we calculate speedup and efficiency with respect to eight workers. For strong scaling, we fix the total number of evaluations at 512 and increase the number of workers, i.e., GPUs. We average over three runs with different seeds and keep four workers per island while increasing the number of islands. Speedup increases up to 128 workers, where we reach approximately half the optimal value. This is an expected decline since each worker only processes few individuals, so the variance in evaluation times leads to larger idle times of the faster workers before the final population synchronization at the end. Additionally, as the number of workers approaches the total number of evaluations, the randomly initialized evolutionary search in turn approaches a random search. This means that the search performance is likely to be worse than what the pure compute performance might suggest. It is still possible to apply Propulate on these scales, but the other search parameters have to be adjusted accordingly as shown in the weak scaling plot (see Fig. 5 top). The early super-scalar behavior is likely due to the non-sequential baseline. For small node counts, the performance is influenced by effects stemming from cluster utilization beyond the use case studied here, like file system congestion or inter-node distance in the network. With larger node

counts relative to total cluster size, these effects average out or approach the worst case, which is consistent with the trend shown in Fig. 5. Weak efficiency only drops to 95% on average at our largest configuration of 128 workers.

Table 4. HP search space of ResNet-50 for BigEarthNet classification.

Optimizers	Optimizer parameters		LR warmup parameters	
Adagrad	Initial accum. value	$[10^{-4}, 0.5]$	LR warmup steps	$[10^0, 10^4]$
SGD	Clipnorm	$[-1, -1000]$	Initial LR	$[10^{-5}, 10^{-1}]$
Adadelta	Clipvalue	$[-1, 1000]$	Decay steps	$[10^2, 10^5]$
RMSprop	Use EMA	Boolean	LR warmup power	$[10^{-1}, 10^1]$
	EMA momentum	$[0.5, 1.0]$		
	EMA overwrite	$[1, 10^3]$		
	Momentum	$[0.0, 1.0]$		
	Nesterov	Boolean		
	Rho	$[0.8, 0.99999]$		
	Epsilon	$[10^{-9}, 10^{-4}]$		

Loss functions			LR parameters	
Binary CE	Categorical CE	Categorical hinge	Decay rate	$[0.8, 0.9999]$
Hinge	KL divergence	Squared hinge	Staircase inverse time decay	Boolean
Activation functions			Decay rate	$[0.1, 0.9]$
ELU	ReLU	Softplus	Staircase polynomial decay	Boolean
Exponential	SELU	Softsign		
Hard sigmoid	Sigmoid	Swish	End LR	$[10^{-4}, 10^{-2}]$
Linear	Softmax	Tanh	Power	$[0.5, 2.5]$

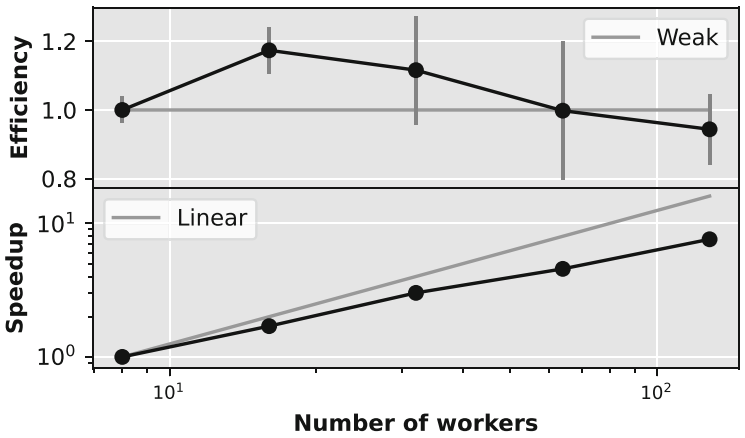

Fig. 5. Scaling with respect to a baseline of eight workers. Weak efficiency (top) and strong linear speedup (bottom). Use case and search space are described in Sect. 4.5. Weak-scaling problem size is varied via the number of OF evaluations. Results are averaged over three runs.

5 Conclusion

We presented Propulate, our HPC-adapted, asynchronous genetic optimization algorithm and software. Our experimental evaluation shows that the fully asynchronous evaluation, propagation, and migration enable a highly efficient and parallelizable genetic optimization. To our knowledge, all existing Python-based genetic optimization tools use synchronization schemes that are not tailored to application in HPC environments. Harder to quantify than performance but very important is ease of use. Especially for HPC applications at scale, some parallelization and distribution models are more suited than others. A purely MPI-based implementation as in Propulate is not only extremely efficient for highly parallel and communication-intensive algorithms but also easy to set up and maintain, since the required infrastructure is commonly available on HPC systems. This is not the case for any of the other tools investigated, except for the not publicly available `MENNDL`. In addition, Propulate's asynchronicity facilitates a tighter coupling of individuals during the optimization, which enables a more efficient evaluation of candidates and in particular early stopping informed by previously evaluated individuals in the NAS case. Propulate was already successfully applied to HPO for various ML models on different HPC machines [13,17]. Another avenue for future work is including variable-length gene descriptions. Mutually exclusive genes of different lengths, such as the parameter sets for Adam- and SGD-like optimizers in our NAS use case, can thus be explored efficiently. While this is already possible, it requires an inconvenient workaround of including inactive genes and adapting the propagators to manually prevent the evaluation of many individuals differing only in inactive genes.

Acknowledgments. This work is supported by the Helmholtz AI platform grant and the Helmholtz Association Initiative and Networking Fund on the HAICORE@KIT partition.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283 (2016)
2. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: a next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2623–2631 (2019). https://doi.org/10.1145/3292500.3330701
3. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002). https://doi.org/10.1109/TEVC.2002.800880
4. Alba, E., Troya, J.M.: A survey of parallel distributed genetic algorithms. Complexity **4**(4), 31–52 (1999)
5. The GPyOpt authors: GPyOpt: A Bayesian Optimization Framework in Python (2016). https://github.com/SheffieldML/GPyOpt

6. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**(10), 281–305 (2012). https://jmlr.org/papers/v13/bergstra12a.html
7. Bergstra, J., Yamins, D., Cox, D.: Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In: International Conference on Machine Learning, pp. 115–123. PMLR (2013). https://proceedings.mlr.press/v28/bergstra13.pdf
8. Bianchi, L., Dorigo, M., Gambardella, L.M., Gutjahr, W.J.: A survey on metaheuristics for stochastic combinatorial optimization. Nat. Comput. **8**(2), 239–287 (2009). https://doi.org/10.1007/s11047-008-9098-4
9. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. (CSUR) **35**(3), 268–308 (2003). https://doi.org/10.1145/937503.937505
10. Bossard, M., Feranec, J., Otahel, J., et al.: CORINE land cover technical guide - Addendum 2000, vol. 40. European Environment Agency Copenhagen (2000)
11. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms, vol. 1. Springer Science & Business Media, Berlin, Heidelberg (2000). https://doi.org/10.1007/978-1-4615-4369-5
12. Cantú-Paz, E., et al.: A survey of parallel genetic algorithms. Calculateurs paralleles, reseaux et systems repartis **10**(2), 141–171 (1998)
13. Coquelin, D., Sedona, R., Riedel, M., Götz, M.: Evolutionary optimization of neural architectures in remote sensing classification problems. In: 2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS, pp. 1587–1590. IEEE (2021). https://doi.org/10.1109/IGARSS47720.2021.9554309
14. Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: a survey. J. Mach. Learn. Res. **20**(1), 1997–2017 (2019)
15. Feurer, M., Hutter, F.: Hyperparameter optimization. In: Hutter, F., Kotthoff, L., Vanschoren, J. (eds.) Automated Machine Learning. TSSCML, pp. 3–33. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05318-5_1
16. Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: DEAP: evolutionary algorithms made easy. J. Mach. Learn. Res. **13**(1), 2171–2175 (2012)
17. Funk, Y., Götz, M., Anzt, H.: Prediction of optimal solvers for sparse linear systems using deep learning. In: Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing, pp. 14–24. Society for Industrial and Applied Mathematics (2022). https://doi.org/10.1137/1.9781611977141.2
18. George, J., et al.: A Scalable and Cloud-Native Hyperparameter Tuning System (2020). https://doi.org/10.48550/arXiv.2006.02085
19. Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., Sculley, D.: Google Vizier: a service for black-box optimization. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1487–1495 (2017). https://doi.org/10.1145/3097983.3098043
20. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
21. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9908, pp. 630–645. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46493-0_38
22. Hertel, L., Collado, J., Sadowski, P., Baldi, P.: Sherpa: hyperparameter optimization for machine learning models. In: 32nd Conference on Neural Information Processing Systems (NIPS 2018) (2018). https://github.com/sherpa-ai/sherpa

23. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press, Cambridge (1992). https://doi.org/10.7551/MITPRESS/1090.001.0001
24. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40
25. Koch, P., Golovidov, O., Gardner, S., Wujek, B., Griffin, J., Xu, Y.: Autotune: a derivative-free optimization framework for hyperparameter tuning. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 443–452 (2018). https://doi.org/10.1145/3219819.3219837
26. Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I.: Tune: a research platform for distributed model selection and training (2018). arXiv preprint arXiv:1807.05118
27. Lindauer, M., et al.: SMAC3: a versatile Bayesian optimization package for hyperparameter optimization. J. Mach. Learn. Res. **23**, 54–1 (2022)
28. Lunacek, M., Whitley, D., Sutton, A.: The impact of global structure on search. In: Rudolph, G., Jansen, T., Beume, N., Lucas, S., Poloni, C. (eds.) PPSN 2008. LNCS, vol. 5199, pp. 498–507. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87700-4_50
29. Luque, G., Alba, E.: Parallel Genetic Algorithms: Theory and Real World Applications, vol. 367. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22084-5
30. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
31. Rapin, J., Teytaud, O.: Nevergrad - A Gradient-free Optimization Platform (2018). https://github.com/FacebookResearch/Nevergrad
32. Snoek, J., Larochelle, H., Adams, R.P.: Practical Bayesian optimization of machine learning algorithms. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) Advances in Neural Information Processing Systems, vol. 25. Curran Associates, Inc. (2012). https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf
33. Song, X., Perel, S., Lee, C., Kochanski, G., Golovin, D.: Open source Vizier: distributed infrastructure and API for reliable and flexible blackbox Optimization. In: Automated Machine Learning Conference, Systems Track (AutoML-Conf Systems) (2022). https://github.com/google/vizier
34. Sudholt, D.: Parallel evolutionary algorithms. In: Kacprzyk, J., Pedrycz, W. (eds.) Springer Handbook of Computational Intelligence, pp. 929–959. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-43505-2_46
35. Sumbul, G., et al.: BigEarthNet Dataset with a New Class-Nomenclature for Remote Sensing Image Understanding (2020). arXiv preprint arXiv:2001.06372
36. Toklu, N.E., Atkinson, T., Micka, V., Srivastava, R.K.: EvoTorch: advanced evolutionary computation library built directly on top of PyTorch, created at NNAISENSE (2022). https://github.com/nnaisense/evotorch
37. Tomassini, M.: Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/3-540-29938-6
38. Wang, J., Clark, S.C., Liu, E., Frazier, P.I.: Parallel Bayesian global optimization of expensive functions. Oper. Res. **68**(6), 1850–1865 (2020). https://doi.org/10.1287/opre.2019.1966

39. Weiel, M., Götz, M., Klein, A., Coquelin, D., Floca, R., Schug, A.: Dynamic particle swarm optimization of biomolecular simulation parameters with flexible objective functions. Nat. Mach. Intell. **3**(8), 727–734 (2021). https://doi.org/10.1038/s42256-021-00366-3
40. Young, S.R., Rose, D.C., Karnowski, T.P., Lim, S.H., Patton, R.M.: Optimizing deep learning hyper-parameters through an evolutionary algorithm. In: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, pp. 1–5 (2015). https://doi.org/10.1145/2834892.2834896