# Gaggle: Genetic Algorithms on the GPU using PyTorch

Lucas Fenaux
lucas.fenaux@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Thomas Humphries
thomas.humphries@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Florian Kerschbaum
florian.kerschbaum@uwaterloo.ca
University of Waterloo
Waterloo, Canada

## ABSTRACT

PyTorch has profoundly impacted the machine learning (ML) community by allowing researchers of all backgrounds to train models efficiently. While PyTorch is the de facto standard in ML, the evolutionary algorithms (EA) community instead relies on many different libraries, each with low adoption in practice. In an effort to provide a standardized library for EA, packages like LEAP and PyGAD have been developed. However, these libraries fall short in either scalability or usability. In particular, neither of these packages offers efficient support for neuroevolutionary tasks. We argue that the best way to develop a PyTorch-like library for EAs is to build on the already solid foundation of PyTorch itself. We present Gaggle, an efficient PyTorch-based EA library that better supports GPU-based tasks like neuroevolution while maintaining the efficiency of CPU-based problems. We evaluate Gaggle on various problems and find statistically significant improvements in runtime over prior work on problems like training neural networks. In addition to efficiency, Gaggle provides a simple single-line interface making it accessible to beginners and a more customizable research interface with detailed configuration files to better support the EA research community.

## CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Computing methodologies → Genetic algorithms**.

## KEYWORDS

Genetic Algorithms, PyTorch, Usable Software

**ACM Reference Format:**

Lucas Fenaux, Thomas Humphries, and Florian Kerschbaum. 2023. Gaggle: Genetic Algorithms on the GPU using PyTorch. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583133.3596356

## 1 INTRODUCTION

An essential tool for rapid research development is an efficient and flexible software library to evaluate new algorithms. PyTorch [5] is the most prominent example of how a software library can have a profound effect on the research community. PyTorch allows for the rapid prototyping of machine learning algorithms with an extensive library of highly optimized tensor-based operations. As faster GPUs become more accessible, PyTorch enables users of all skill sets to train large models in minutes on the GPU. PyTorch is also highly portable due to the high demand for supporting new architectures.

A recent line of work from the evolutionary algorithm (EA) community has focused on designing a tool with the same modularity and usability of PyTorch [3, 9]. This line of work is motivated by the many disjoint implementations of EAs with low adoption in practice. One of the most prominent works that aims to be the PyTorch of EAs is LEAP [3]. LEAP is a highly optimized CPU-based EA library with a functional programming design. LEAP allows researchers to easily modify the EA pipeline by swapping evolutionary operators, taking measurements, and deploying their work to high-performance computing architectures. Despite LEAP's modular design, implementing new problems is challenging due to the many layers of abstraction within the package. Specifically, we found implementing problems like neuroevolution (where LEAP and PyTorch need to work together) particularly difficult.

The most widely used EA software library is PyGAD, with nearly a million pip installs [6]. The greatest strength of PyGAD is that it has allowed users with little to no background to solve problems and gain experience with EAs. The most significant drawback of PyGAD is its scalability, as demonstrated in our evaluation. Furthermore, it is hard to modify due to its reliance on global variables and lack of modularity. However, PyGAD does support neuroevolutionary tasks using a PyTorch wrapper. This wrapper converts PyTorch models to Numpy arrays and vice versa, with all EA operations happening in Numpy. The problem with a conversion approach is the overhead of constantly moving ML models on and off the GPU.

Motivated by the strengths and weaknesses of both LEAP and PyGAD, we develop a new software library called Gaggle[1]. Gaggle uses PyTorch as the backend for several reasons. First, we argue the best way to get a PyTorch like library for EAs is to build on the already solid foundation of PyTorch itself. Using PyTorch makes it much easier and faster to support EAs whose fitness function relies on PyTorch models in any way. Our evaluation shows that Gaggle consistently outperforms both LEAP and PyGAD in time per generation with statistical significance (up to 9.7x) when evolving neural networks. Finally, the efficiency of PyTorch allows us to be competitive with LEAP even on CPU-only tasks such as Rastrigin, matching their scalability up to a tenth of a second constant factor.

In addition to the significant performance improvements, we design Gaggle with usability in mind. Inspired by LEAP, we use a modular design making it simple to configure different combinations of evolutionary operators to create different algorithms. Gaggle uses an object-orientated design that makes it easy for researchers to implement new behaviours by inheriting existing

[1] https://github.com/LucasFenaux/torch-gaggle


*GECCO '23 Companion, July 15–19, 2023, Lisbon, Portugal*

ACM ISBN 979-8-4007-0120-7/23/07…$15.00
https://doi.org/10.1145/3583133.3596356

functionality from the built-in objects. Furthermore, we adopt the factory pattern from object-orientated design to allow all aspects of the algorithm to be specified in a configuration file. Configuration files specifying all parameters used in an experiment are an essential tool for reproducibility and efficiency in research experimentation. Finally, inspired by PyGAD, we also maintain a simple interface for beginners by offering a GA supervisor object that gives access to most features of Gaggle in a single line of code.

## 2 DESIGN

### 2.1 Package Overview

Gaggle follows an object-orientated design with a pyramid-like structure. At the top is the GA object, representing the high level type of EA (e.g. simple, steady-state). This object takes all other objects in its constructor and defines how they interact with each other (and when to call each operator). Put another way; the GA object is analogous to the trainer object that can be found in many PyTorch libraries. We describe in more detail the three core components of the GA: the *Population Manager*, *Problem* object and *Operator* objects.

*Population Manager.* The population manager acts as the primary data structure for the GA. It stores and manages all the individuals keeping track of their fitness. To avoid wasteful computation, we optionally keep track of an individual's *freshness* and only recompute its fitness if it was modified since its fitness was last computed. The Population Manager also provides a standardized interface for the evolutionary operators to access and update individuals. It stores all the individual-related meta-information required for the operators, such as which parents have been chosen, which individuals will be crossed over etc. This simplifies the operator pipeline by avoiding the need for operators to interface with each other directly.

*Operators.* Operators are highly cohesive objects representing the basic operations that make a GA: selection, crossover, and mutation. We include a selection of the most common operators, such as uniform crossover, roulette wheel selection and Gaussian mutation. Due to Gaggle's PyTorch backend, it is also simple and efficient to add more complex operators, such as SGD, for a local improvement operator to create a hybrid GA.

*Problem.* A problem object encompasses everything about the fitness function the GA optimizes. This can be a simple benchmark function such as Rastrigin, or more complex problems such as neural network training and reinforcement learning. The problem object's main purpose is to evaluate an individual and return its fitness to the population manager. The problem object can also store objects like datasets, PyTorch models, and RL environments. We also provide direct support for LEAP problems and the OpenAI Gym suite for RL problems.

*Customization.* Due to Gaggle's high-cohesion, low-coupling object-oriented design, it is straightforward for researchers to modify all parts of the code with little to no downstream effects. Registering these modifications in Gaggle's factories takes a single line (see Listing 1 for an example). Once registered, Gaggle allows custom code to be invoked using its pre-built configuration file system for simple and reproducible research code.

```
    # creating the problem
    class MaxOnesProblem(Problem):
        def evaluate(self, individual: Individual, *args,
      **kwargs) -> float:
            chromosome = individual()
            return torch.sum(chromosome).cpu().item()
    #register the problem in the factory
    ProblemFactory.register_problem(
        problem_type='custom', problem_name='maxones',
     problem=MaxOnesProblem)
    # parse args
    outdir_args, sys_args, individual_args, problem_args,
      ga_args, config_args = parse_args()
    # if config file is specified, overwrite arguments
    if config_args.exists():
        outdir_args, sys_args, individual_args,
     problem_args, ga_args = config_args.get_args()
    # run
    trainer: GA = GAFactory.from_ga_args(ga_args=ga_args,
      problem_args=problem_args, sys_args=sys_args,
     outdir_args=outdir_args, individual_args=
     individual_args)
    trainer.train()
```

**Listing 1: Creating and registering a custom problem (MaxOnes) to be used with our configuration file system.**

```
GASupervisor(problem_name="MNIST", individual_name="nn",
             model_name="lenet", device="cuda").run()
```

**Listing 2: Example of using the GA Supervisor for a pre-built problem**

```
def fitness_function(individual):
    chromosome = individual()
    genome_size = individual.get_genome_size()
    rastrigin = - (10 * genome_size + torch.sum(
     chromosome ** 2 - 10 * torch.cos(2 * torch.pi *
     chromosome)))
    return rastrigin.cpu().item()

supervisor = GASupervisor(individual_name="pytorch",
                          individual_size=100)
supervisor.set_custom_fitness(fitness_function)
supervisor.run()
```

**Listing 3: Example of using the GA Supervisor for a custom fitness function (Rastrigin)**

### 2.2 Configuring the GA

*GA supervisor.* To facilitate fast implementation and to make Gaggle accessible to beginners, we include a GA supervisor class. This class gives a single-line interface to our system. At minimum, a user needs to specify which problem they want to solve and some arguments, as shown in Listing 2. However, we also allow for customization by passing functions or new operators. For example, if we want to code the Rastrigin problem, we could pass a custom function as in Listing 3.

*Research Mode.* The default usage of our code is research mode, which uses a similar design to PyTorch, where one assembles a GA by initializing the various objects and connecting them. One of the

most important differences between this and the GA supervisor is the ability to use config files for reproducibility and organization of experiment code. Our argument parser allows for either command line arguments or a standard yml file. We give an example usage that includes defining a custom problem in Listing 1.

## 3 RELATED WORK

We focus our evaluation on the LEAP [3] and PyGAD [6] libraries. LEAP is the current state-of-the-art Python library for features and usability, while PyGAD is the most popular, with almost a million pip installs. We defer to the LEAP paper for a complete list of previous Python libraries. We are the first library to focus on both usability and GPU acceleration. JEGA is a new Java-based framework with many design goals in common with our work, including speed, modularity, and portability [9]. However, using PyTorch allows us to better meet these design goals while offering seamless support of neuroevolution.

Speeding up evolutionary algorithms using the GPU is not a new idea. Cheng and Gen give an extensive survey of various attempts [2]. Gaggle complements these works by making it easier to utilize the GPU for several reasons. First, Gaggle's PyTorch backend makes using the GPU as simple as setting the device (assuming the operations would benefit from the GPU). Using PyTorch also gives flexible deployment options to various architectures and access to extensive documentation from the wider ML community. Klosko et al. recently proposed accelerating evolutionary operators by making them more amenable to tensors [7]. Gaggle complements this work by offering a user-friendly tensor-based EA implementation upon which Klosko et al.'s operators can easily be implemented. We focus on the initial package in this work, but adding further optimizations such as Klosko et al.'s is an important future work.

## 4 EVALUATION

### 4.1 Evaluation Setup

This section provides a series of micro-benchmarks against the LEAP and PyGAD libraries [3, 6]. We focus on time-per-generation as the metric to compare each library. To accurately measure the run time, we fix all hyperparameters and operators so that each library is running the exact same GA. Specifically, we use roulette wheel selection, uniform crossover (with 50% crossover probability), and uniform mutation (with 1% probability). Unless otherwise specified, we run a simple generational GA for 100 generations with a population size of 100. We note that these hyperparameters are not optimal as fitness is not a metric we are interested in. We assume an EA giving better utility could be implemented in any of the libraries (with varying degrees of difficulty).

All evaluations were run on a machine with 1/4 TB of RAM and an 8-core Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz and a Tesla P100 GPU with 12GB of memory. We run the GA for 100 generations for each configuration and report the mean time-per-generation along with a 95% confidence interval represented as a shaded area (a very small region in most plots as our timing is quite consistent). We make our benchmark code available to reproduce the results[2].

[2]https://github.com/LucasFenaux/gaggle-benchmarking

### 4.2 Supervised Learning

*4.2.1 Problem Description.* We consider the problem of training a neural network (NN) using a GA by evolving the weights of the model. We use a LeNet [8] model with 61.7k parameters. The training dataset is MNIST [4], a classic benchmark for supervised learning where the classification problem is identifying handwritten digits from 0-9. A chromosome is a set of parameters for the NN. The fitness function is the accuracy of the NN on the training dataset. In this experiment, we fix the NN architecture and all hyperparameters and only vary the population size of the GA to show how each library scales. PyGAD supports NN training using two helper functions that convert PyTorch models to Numpy arrays and vice versa. We use these helper functions to support both PyGAD and LEAP for NN training. The forward pass of the NN is done in PyTorch and thus can be done on the GPU. We allow both LEAP and PyGAD to use the GPU for inference (although both use the CPU for all GA operations).

*4.2.2 Results.* We give the results in Figure 1. On the x-axis is the size of the population, and on the y-axis is the time per generation in seconds. We find that for all population sizes, Gaggle outperforms both PyGAD and LEAP with statistical significance, as the 95% confidence intervals do not overlap (the very small shaded area around each line). Furthermore, Gaggle scales much better to larger populations. This is due to the fact that Gaggle keeps all models and data on the GPU. Conversely, PyGAD and LEAP need to bring all models off the GPU and convert them to Numpy arrays in order to perform GA operations, then re-initialize the PyTorch models for the next generation. Gaggle instead modifies the PyTorch models directly on the GPU when conducting GA operations. We note that the GPU we use is small by today's standards. We also ran this experiment on a NVIDA A100 with 80GB of RAM. We observed a similar plot to Figure 1, with the difference between Gaggle and LEAP being even more significant.

### 4.3 Reinforcement Learning

*4.3.1 Problem Description.* Genetic Algorithms can provide a competitive alternative to gradient-based techniques for training reinforcement learning agents. We consider the popular Cartpole benchmark, where the goal is to train an agent to balance a vertical pole on a moving cart. We use Open AI's Gym [1] simulation for this problem (recall that Gaggle supports any OpenAIGym problem). A chromosome is a set of parameters for the policy model. The fitness function is the reward of the policy in the Cartpole environment. In this experiment, we fix all hyperparameters (including the population size to 100) and only vary the number of learnable parameters in the policy model. For a fair comparison, we set `stop_on_done` to False in the gym environment (so that every run has the same number of steps).

We use the same neural network model as LEAP for our smallest policy model (30 parameters). To show how the different framework's scale, we progressively add more parameters by increasing the size of the layers and the number of layers from 2 to 3. This yields five models with 30, 114, 450, 17410, and 67590 trainable parameters. The smallest model is included as an example problem in LEAP (using Numpy only). We implement all models in PyTorch for both Gaggle and PyGAD. The smallest model was trained on

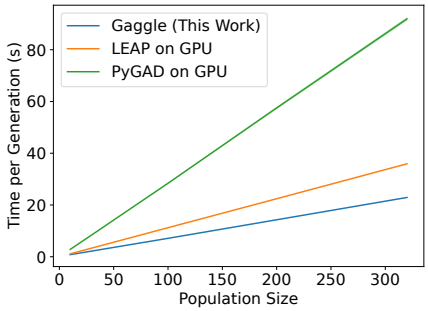


**Figure 1: MNIST NN training evaluation**

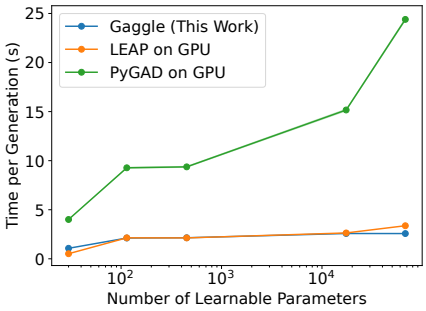


**Figure 2: Cartpole RL evaluation**

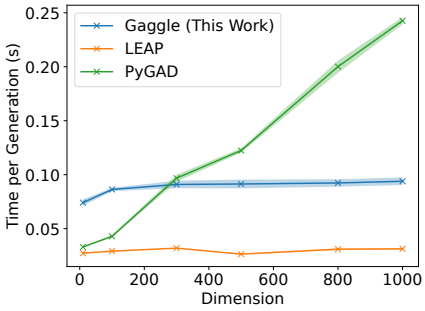


**Figure 3: Rastrigin evaluation**

the CPU for all libraries to ensure a fair comparison with LEAP. All other models used the GPU. For LEAP, we had to create a custom decoder that uses PyGAD's conversion interface to transform the larger models into a flattened Numpy array.

*4.3.2 Results.* We give the results in Figure 2. On the x-axis is the number of learnable parameters in the policy models, and on the y-axis is the time per generation in seconds. Gaggle performs competitively with LEAP, scaling better to larger models. This is again due to its GPU-centric design that avoids unnecessary model conversions. PyGAD is significantly outperformed in all settings by both LEAP and Gaggle. We remark that Gaggle's advantage is less significant than in supervised learning. This is because the Cartpole is an easy problem to implement and solve. Thus, it does not make sense to use deeper neural networks. However, for more complex reinforcement learning problems that require significantly larger models, Gaggle's advantage would be significantly more pronounced.

## 4.4 Benchmark Function

*4.4.1 Problem Description.* Finally, we consider a real-valued benchmark problem to show how we perform on non-machine learning problems. We choose the Rastrigin Benchmark function, defined as

$$F(\vec{x}) = 10d + \sum_{i=1}^{d} \left( x_i^2 - 10\cos(2\pi x_i) \right) \quad (1)$$

where the chromosome $\vec{x}$ is a vector of dimension $d$ with each $x_i \in [-5.12, 5.12]$ (and $F$ is the fitness). We fix all hyperparameters and vary only the dimension $d$ to show how the various libraries scale. This problem is already implemented in the LEAP library, although we observed it actually maximizes the function. We add a negative sign to the Rastrigin function to fix this problem. For PyGAD, we pass LEAP's evaluate function as the fitness. Finally, we code the Rastrigin function in PyTorch for Gaggle (although Gaggle can also support any LEAP problem).

*4.4.2 Results.* We give the results in Figure 3. We find that Gaggle and LEAP have a similar slope as $d$ increases showing that PyTorch (Gaggle) and Numpy (LEAP) scale similarly to more complex problems. However, Gaggle incurs a larger overhead using PyTorch, which is constant and less than a tenth of a second. However, PyGAD scales linearly with the dimension of the problem.

# 5 CONCLUSION

We have shown that Gaggle can build upon the success of PyTorch to offer a scalable solution to machine learning-based problems in EAs. We achieve state-of-the-art run times on a variety of problems, allowing EAs to scale to deeper models and larger populations than previous libraries. Gaggle makes it simple for researchers to design, configure, deploy, and measure new research ideas through an easy-to-modify object-orientated design. We achieve this without sacrificing usability for novice users by also providing a one-line interface. Thus, in time, Gaggle aims to enable faster research and unify the GA community as PyTorch did the ML community.

# ACKNOWLEDGEMENTS

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, IRC-537591, the NSERC Postgraduate Scholarship-Doctoral program, and the Royal Bank of Canada for funding this research.

# REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. https://doi.org/10.48550/arXiv.1606.01540 arXiv:arXiv:1606.01540

[2] John Runwei Cheng and Mitsuo Gen. 2019. Accelerating Genetic Algorithms with GPU Computing: A Selective Overview. *Computers & Industrial Engineering* 128 (Feb. 2019), 514–525. https://doi.org/10.1016/j.cie.2018.12.067

[3] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. 2020. Library for Evolutionary Algorithms in Python (LEAP). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1571–1579. https://doi.org/10.1145/3377929.3398147

[4] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* 29, 6 (Nov. 2012), 141–142. https://doi.org/10.1109/MSP.2012.2211477

[5] Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[6] Ahmed Fawzy Gad. 2021. PyGAD: An Intuitive Genetic Algorithm Python Library. https://doi.org/10.48550/arXiv.2106.06158 arXiv:arXiv:2106.06158

[7] Jonatan Kłosko, Mateusz Benecki, Grzegorz Wcisło, Jacek Dajda, and Wojciech Turek. 2022. High Performance Evolutionary Computation with Tensor-Based Acceleration. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 805–813. https://doi.org/10.1145/3512290.3528753

[8] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. https://doi.org/10.1109/5.726791

[9] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. 2022. JGEA: A Modular Java Framework for Experimenting with Evolutionary Computation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 2009–2018. https://doi.org/10.1145/3520304.3533960