An Oversimplified look on the Gradient Population Optimization for Large-Scale Search

Pablo Acereda
Computer Science Degree

March 24, 2019

Abstract

Rewarding the content of this file, it will be found an oversimplification of the content of the article [6], as the objective of this assignment is to grasp an article format and writing. Thus, the content will not have as much detail as found at article [6]. Nevertheless, the article presents the algorithm Gradient Population Optimization (GPO), which is a novel scalable algorithm with designed for the optimization of cost functions. It uses the TensorFlow platform.

Keywords: Evolutionary Computation, Particle Swarm Optimization, Parallel Architectures, Gradient Methods.

1 Introduction

Optimization can be used in a wide of study fields, and using a large variety of algorithms (Bayesian networks [5], genetic reproduction [2], gradients [4], evolutionary populations [3], . . .).

One of the main problems faced when optimizing a certain function are confronted in large-scale optimizations, due to the *high-dimensionality* and the *non-linearity*. That's one of the reasons why global search is used.

High-dimensionality is one of the most urgent barriers against scalability for machine learning and engineering algorithms, thus, the *Gradient Population Optimization* (GPO) has been developed to try to improve the current scalability problem.

The presented algorithm uses a *dataflow graph* (DFG) paradigm, which involves a non-Von-Neumann processing framework (recently used for high-performance computing research). It has been implemented in the framework known as *TensorFlow* (TF). Using a hybrid implementation of a local and a global search mechanisms it is intended to surpass current searching algorithms.

The standard algorithm PSO already implements a local and a global search mechanism, but here are the changes introduced in order to improve PSO performance:

1. A gradient operator to replace PSO social operator.
2. An interaction is introduced between the local and global position update operators.

Gradient operation where "computationally expensive" until more *symbolic* implementations appeared within frameworks of dataflow graphs (Caffe and TensorFlow).

The logic behind the first statement was, for the authors, to think the result of the search space after it becoming too large: it becomes "continuum". Therefore, rendering a gradient operator in the thing to be done taking that it is more efficient in discerning the rich information hidden in the continuum immediately surrounding of each particle.

The second innovation is the instantiation of a coupling mechanism between the local and the global operator, implemented by a *gradient clipping operation*.

2 Non-Von-Neumann Computing: The dataflow model

The authors claim that a parallel system needs to get rid of a Von-Neumann paradigm, as its sequential execution does not fit dataflow problems execution. They give several facts to support the statement:

1. The main part of the program are not the instructions, but *data* is. The system is based on how data flows and interacts with other data streams.
2. Since data flows in real time, no global memory is needed anymore.
3. No *Program Counter* (PC) is needed since data processing is executed only when local conditions are satisfied.

For this paradigm implementation, the authors focused on heterogeneous implementation (usage of CPU and GPU). Thus, TensorFlow-based dataflow codes does not require substantial reprogramming according to the writers.

3 Heterogeneous Computing

Almost any house-computer currently being used comes with a CPU and a GPU integrated. Intuitively, anyone with basic knowledge about computing would think that increasing the number of cores in a CPU would increase the processing power of the device, at it would allow more parallel tasks at the same time. But, on the contrary, current CPUs tend to increase clock speed and other specifications instead of the number of cores. That obviously increases the processing power per thread, but it does not mean it increases parallelism. And that is where GPU comes in handy, with many cores (although less powerful than the CPU cores), to process and render parallel operations.

Having said that, the GPUs purposes have increased in the late decades, as they allow that parallel computation (training artificial neural networks and running optimization algorithms).

Therefore, CPUs are used when the calculations cannot be run in parallel, as they have superior clock frequencies and more low level memory cache (single threaded sequential performance) 1.

Actually, five of the top hundred supercomputers have already implemented some for of heterogeneous computing [1].

4 Dataflow graphs

A computational graph, other name given to a dataflow graph, is a directed graph where the nodes describe operations and edges represent data flowing between these operations. Each of those nodes count with zero or more inputs and zero or more outputs for the values to flow between nodes (which are represented by tensors - arrays with arbitrary dimensionality).

TensorFlow actually supports a distributed implementation where client and processes are run in entirely different machines.

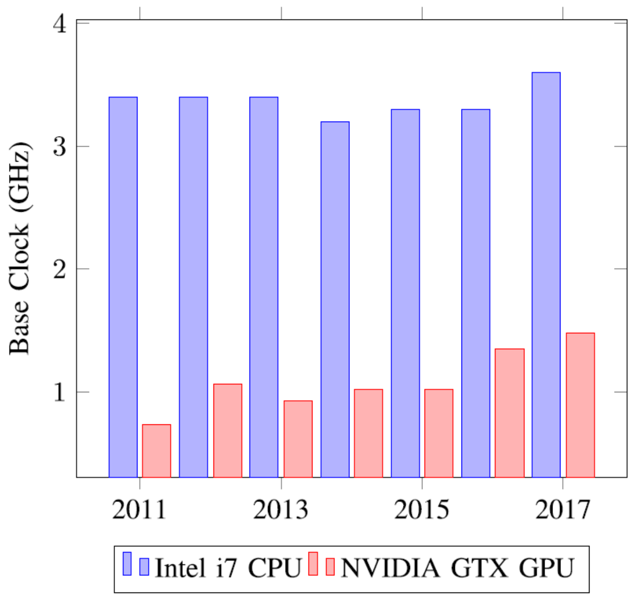

Figure 1: Base clock frequencies of Intel i7 CPU vs. NVIDIA GTX CPU since 2011.

Furthermore, in TensorFlow all data is represented by tensors, independently of their size or complexity. After having the constructed the graph depending on the data and functions used, the simulation is run by a greedy heuristic algorithm.

5 Particle Swarm Optimization

5.1 The classical PSO Algorithm

The PSO algorithm (Particle Swarm Optimization) is an evolutionary algorithm which includes global and local search mechanisms. It simulates a swarm of particles [3]. Each volume-less particle occupies a point in the domain of the cost function as a set of parameters with length *n*. The amount of particles *m* is usually chosen based on the scale or complexity of the cost function [6].

$$P = [x_{i,j} \in R^{m\times n} | x_{i,j} \sim U([l_{min}, l_{max}])] \tag{1}$$

As this is a summary of the main article, it is not going to be specified more equations used in the PSO algorithm. For more information please reference article [6].

5.2 Modifications and Variants

PSO has suffered along the years several meta-optimizations. In the variant used in the summarized paper the previous velocity in the velocity calculation is scaled by an inertia value ω.

$$v_i := \omega v_i + \varphi_1 s_1(\vec{b_i} - \vec{p_i}) + \varphi_2 s_2(\vec{g} - \vec{p_i}) \tag{2}$$

Therefore, linearly decreasing inertia values tend to increase performance of the global and local searches. This leads to a problem, which is that the algorithm does not perform well the it has not been restricted by local information.

It was found that combining PSO and genetic algorithms leads to better results than those algorithms separately.

5.3 Cognitive limitations

Basically, the algorithm has problems while operating in high dimensions functions.

5.4 Transition from PSO to GPO

The main difference between these algorithms will be found of the way the local and global search mechanisms interact with one another. In GPO, the local search is going to be affected by the gradient operator.

The constant interaction from the global search providing information to the local search gives the algorithm the ability to coordinate both explorations.

6 The gradient population optimization algorithm

The GPO algorithm seeks to exploit the advantage of the GD search for the local mechanism; and the advantage of the PSO for the global mechanism.

6.1 GPO algorithm: the fundamental formulation

The GPO algorithm uses the following formula:

$$P := \omega P + F_l(P) + F_h(P) \tag{3}$$

In the previous formula, ωP represents the inertia factor; while F_g and F_l represents the global and local update factors. The GPO algorithm leaves F_g unchanged, while rewrites F_l.

As for the gradient matrix, it is produced via

$$G = \nabla f(P) \tag{4}$$

Where each row of G is the first order derivative of the corresponding position in the vector in P:

$$Gij = \frac{\partial f(x_i)}{\partial x_j}, i = 1, 2, \dots, m; j = 1, 2, \dots, n \tag{5}$$

In TensorFlow the gradient expression 5 is computed using special highly-efficient symbolic/numeric routine that makes extensive use of the graph-like structure of the dataflow implementation of the algorithm.

According to the authors, the formula 4 cannot be applied in practice. The gradient needs to be controlled in order to avoid the algorithm to explode while updating the evolution.

To do so, authors performed a *two-step clipping procedure*.

1. G is clipped by its direct global norm.

$$\|G\|_{Fr} = (\sum_{i=1}^{m}\sum_{j=1}^{n}|G_{i,j}|^2)^{\frac{1}{2}} \tag{6}$$

Next, the Frobenius norm $\|G\|_{Fr}$ is used to determine the following scaling factor

$$\alpha = \frac{\gamma}{max(\|G\|_{Fr}, \gamma)} \tag{7}$$

After some experimentation, $\gamma = 2.5$ was the selected value for 7 equation.

Finally, the α-factor is applied to the matrix G through the scalar multiplication:

$$G' = G\alpha \tag{8}$$

2. Afterwards, G' is coupled with F_g obtaining $G' - F_g$. What it does it to clip the first factor by the global optimization best matrix (which is the second factor) only if the next condition is fulfilled:

$$|F_{gi,j}| < |G'_{i,j}| \tag{9}$$

The final update is carried by:

$$F_l = \varphi_l[x_{i,j} \in G' | x_{i,j} = min(|G'_{i,j}|, |F_{gi,j}|)]. \tag{10}$$

According to the authors *This keeps the local factor in check and on par, in terms of scale, with the relatively tame global factor and makes the use of gradients in non-convex optimization more stable* [6].

7 Metrics

To assure clearly in the measurement taken in each algorithm, the samples were taken every 25 epochs. Each result was explained taking into account the result of each algorithm comparatively and explaining the configuration used to obtain that solution (as this is a summary of the original article, there will be no such detail in the explanations given, for more detail please read the whole article [6]).

In the comparisons, the first tier was made with bi-dimensional functions, the three remaining tiers where multi-dimensional functions.

Each multidimensional function was separated into ranges:

1. $10D, 20Dand30D$ being low-dimension.
2. $100D, 500D, 10^3D$ comprising medium-dimension.
3. $10^4D, 10^5D$ for high-dimension functions.

7.1 Experiment settings

As part of the experiment, it was used a high-performance computer with the specifications:

CPU Intel i7 5820K with the factory-set clock speed of 3.3GHz.

GPU NVIDIA Titan X Pascal with 3584 cores and the factory-set clock speed of 1.5GHz.

RAM 64GB.

Cf.	GPO params.	PSO params.	GD params.
C_1	$\varphi_l = 3, \varphi_g = 2$	$\varphi_1 = 3, \varphi_2 = 2$	$\alpha = 0.1$
C_2	$\varphi_l, \varphi_g = 2$	$\varphi_1, \varphi_2 = 2$	$\alpha = 0.01$
C_2	$\varphi_l = 2, \varphi_g = 3$	$\varphi_1 = 2, \varphi_2 = 3$	$\alpha = 0.001$

Table 1: Hyper-parameter configurations used for experimentation.

8 Results

The results of the different functions are going to be explained without getting into much detail, as the objective of this article is to summarize the original [6], not to repeat it.

8.1 Two-dimension function results

The GPO and PSO algorithms had no problem with these functions. Even though, PSO needed more iterations than GPO to find the optimum.

8.2 Low-dimension function results

8.2.1 Sphere function in 10, 20 and 30 dimensions

All of the algorithms were able to find the result.

8.2.2 Griewank function in 10, 20 and 30 dimensions

No algorithm found a solution, although GPO and PSO were able to find an approximation. In general, PSO found obtained the best performance.

8.2.3 Rastrigin function in 10, 20 and 30 dimensions

No algorithm found the global optimum. Even though, GPO had the lowest result average, while PSO had the highest. GD was not able to converge on acceptable local optima.

8.2.4 Styblinski-Tang function 10, 20 and 30 dimensions

No algorithm was able to converge on the global optimum. GPO outperformed PSO and GD.

8.3 Mediom-dimension functions

8.3.1 Sphere function in 100, 500 ns 10^3 dimensions

GPO and GD were able to find the global optimum in every configuration. GPO and GD obtained better overall performance than PSO.

8.3.2 Griewank function in 100, 500 and 10^3 dimensions

No algorithm was able to find the global optimum. GPO was the one which approximated the most (asymptotically).

8.3.3 Rastrigin function in 100, 500 and 10^3 dimensions

GPO found the best result in 100D. On average, PSO had the highest result. GD did not provide an acceptable result.

8.3.4 Rosenbrock function in 100, 500 and 10^3 dimensions

GPO was the only algorithm that found the global optimum. PSO gave acceptable results, but had a higher result on average. GD was not able to produce acceptable results.

8.3.5 Styblinski-Tang function in 100, 500 and 10^3 dimensions

No algorithm was able to find the global optimum. However, GD achieved better result int 500D and 10^3D.

8.4 High-dimension functions

8.4.1 Sphere function in 10^4 and 10^5 dimensions

GPO and GD were able to find the global optimum. PSO was not able to converge in the global optimum in any of the samples.

8.4.2 Rastrigin function in 10^4 and 10^5 dimensions

No algorithm was able to approximate the global optimum. GD obtained the best results.

8.4.3 Rosenbrock function in 10^4 and 10^5 dimensions

GPO provided the best results. PSO achieved acceptable results. GD was not able to provide satisfactory results.

8.4.4 Styblinski-Tang function in 10^4 and 10^5 dimensions

Both GD and GPO provided acceptable results. PSO could not ameliorate past the initialization range.

8.5 Algorithm scalability

In general, the PSO algorithm found scalability difficulties as the dimensions increased. On the other hand, GPO did not find problems in great deal of the functions (only has problems with Rastrigin function). Finally, GD, which had no problem in the Sphere and in the Styblinski-Tang functions, but found lots of problems in the rest of the functions.

8.6 Runtime

8.6.1 PSO NumPy and TensorFlow CPU variants

Summarized in the graph 2.

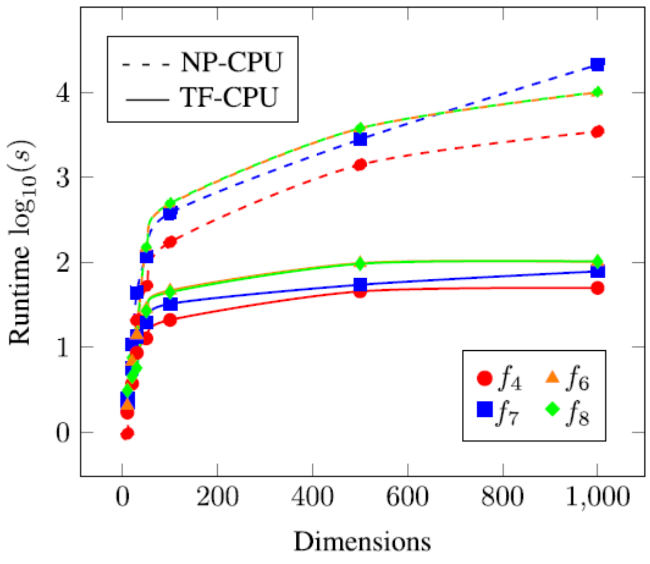

Figure 2: Runtime of PSO variants NP-CPU and TF-CPU on multi-dimension samples.

8.6.2 PSO TensorFlow CPU and GPU variants

Summarized in the graph 3.

8.6.3 GPO and PSO TensorFlow GPU variants

Summarized in the graph 4.

9 Conclusion

According to the authors, the integration of the gradient into the local search mechanism was a success, allowing small populations to optimize exceptionally large problems. Although large populations may cause an increased runtime, or intractable optimization problems.

Overall, GPO obtained good results on most of the tested functions regardless the dimensionality (with less iterations than PSO).

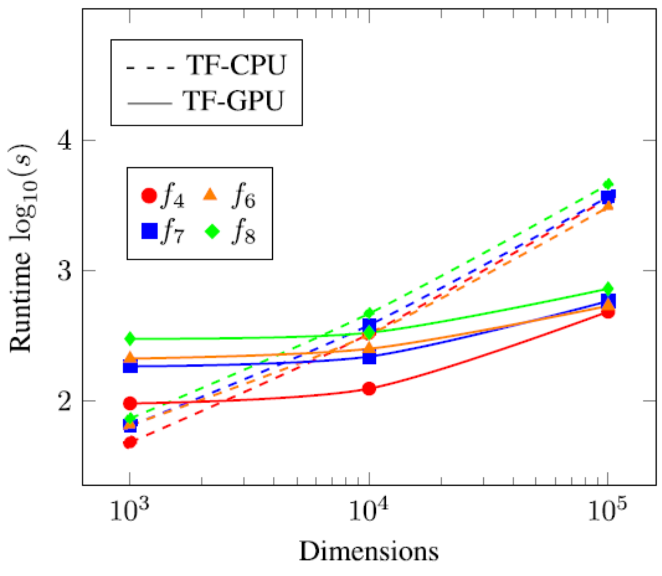

Figure 3: Runtime of PSO variants TF-CPU and TF-GPU on multi-dimension samples.

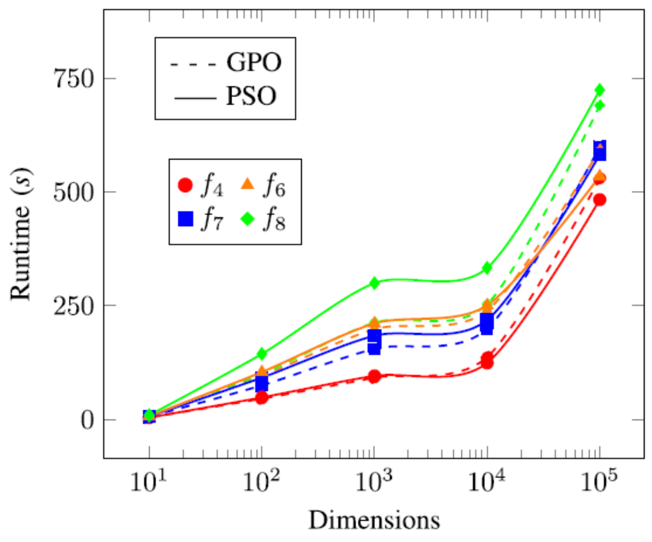

Figure 4: Runtime of GPO and PSO using TF-GPU variants.

Bibliography

References

[1] Pieter Hijma et al. “Cashmere: Heterogeneous Many-Core Computing”. In: *IEEE International Parallel and Distributed Processing Symposium* (May 2015).

[2] Russell Stuart J. and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Third Edition. Pearson Education, 2010.

[3] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *IEEE* (Dec. 1995).

[4] A. Makovetskii and V. Kober. “Analysis of the gradient descent method in problems of the signals and images restoration”. In: *Pattern Recognition and Image Analysis* (Jan. 2015).

[5] Martin Pelikan, D. E. Goldberg, and Tsutsui Shigeyoshi. “Hierarchical Bayesian optimization algorithm: toward a new generation of evolutionary algorithms”. In: *IEEE Journals & Magazines* (Sept. 2003).

[6] John Persano, Said M. Mikki, and Yahia M. M. Antar. “Gradient Population Optimization: A Tensorfloaw-Based Heterogeneous Non-Von-Neumann Paradigm for Large-Scale Searh”. In: *IEEE Journals & Magazines* (July 2018), p. 26.