Simple Particle Swarm Approach for Hyperparameter Optimization

Kaisa Lepajõe

July 31, 2020

1 Introduction

1.1 The Importance of Hyperparameter Optimization

Many disciplines, from Physics to Business, infolve optimization. The optimal solution could be one that minimizes costs and maximises efficiency of a problem. In the realm of machine learning, dimension spaces are large and parameters have special constraints. The goal there is to make an algorithm learn as quickly and effectively as possible. There are many different methods for optimizing parameters. For simple problems, it is feasible to try out parameters manually, or to use random search. More complex problems with large search spaces require sophisticated methods, such as Bayesian or Evolutionary algorithms.

1.2 What is Particle Swarm Optimization

PSO is considered a subset of evolutionary algorithms. It is different from genetic algorithms because instead on focusing on the microscopic DNA-level they focus on the macroscopic level - social structures and behaviors.

Particle Swarm Optimization (PSO) methods are inspired by nature. They mimic flocks of birds, bees or ants. Like an ant colony, each individual has a limited number of responses but the swarm exhibits traits of intelligence as a whole. The dynamics of the swarm is modelled similar to particle physics.

The algorithm created for this project is based on the simplest PSO algorithm introduced in Maurice Clerc's book *Particle Swarm Optimization*.

1.3 Motivation

There are many reasons for choosing Particle Swarm Optimization (PSO) over other traditional optimization algorithms. PSO uses less resources and it can search large spaces of candidate solutions. It does not require the gradient, so the target function does not have to be differentiable or even continuous.

Particle Swarm Optimization has been shown to work in optimizing neural networks. However, it is not guaranteed that an optimal solution will be found - much like in nature.

Another reason for choosing PSO is that there is a benefit to learning about the biology of swarms together with computer science. Interesting things happen where there is a union between two very different fields.

1.4 The Target Problem

The goal of this project is to apply PSO to finding the minimum of a function. We use a widely known test problem - the Rosenbrock banana function.

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2 \tag{1}$$

The known minimum of this function is 0 at the point [1,1]. The difficulty of this problem lies in the shape of the banana function. The global minimum lies in a very flat valley with steep walls. Gradient descent methods do not work well on this problem because the area around the global minimum has a very small gradient.

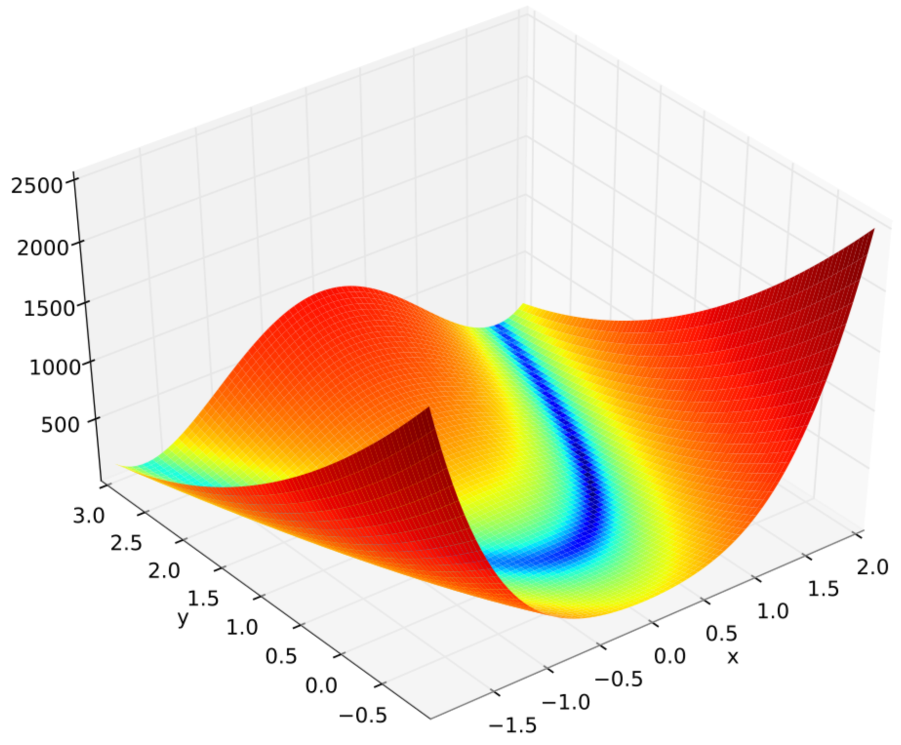

Figure 1: A 3D plot of the Rosenbrock Function

2 The Optimusbeez Package

The PSO algorithm created for this problem was conveniently gathered into a package called optimusbeez. This package is available on PyPi and can easily be installed with the command
>>>pip install optimusbeez

Make sure you install the latest version! At the time of writing, this is 1.0.0.

2.1 How it works

As mentioned previously, the algorithm is based on PSO(0) from Maurice Clerc's book. A swarm of N particles is created. Each particle has k informants. On each time step, the particles evaluate their current position, communicate with their informants, update their velocity and end up in a new positon. The main update equation for the velocity and next position is

$$\begin{cases} v_d \leftarrow c_1 v_d + c_{max}\text{random}(0,1)(p_d - x_d) + c_{max}\text{random}(0,1)(g_d - x_d) \\ x_d \leftarrow x_d + v_d, \end{cases} \tag{2}$$

where d is the dimension, x_d is the particle's position, c_1 and c_{max} are two confidence parameters, p_d is the best position that a particle has visited, and g_d is the best position the particle is aware of. random(0,1) generates a random number from 0 to 1.

v_d is calculated separately for each dimension. It is also constrained to a maximum velocity $v_{max} = \frac{x_{max} - x_{min}}{2}$. The particle is constrained to a rectangular search area given by x_{min} and x_{max}. If the possible next position of the particle is outside the bounds, then the particle is placed at the boundary and the velocity of the particle is set to 0.

In order for the swarm to converge, c_1 and c_{max} must be dependent and follow the relation

$$\begin{cases} c_1 = \frac{1}{\phi - 1 + \sqrt{\phi^2 - 2\phi}} \\ c_{max} = \phi c_1 \end{cases} \tag{3}$$

with another parameter ϕ. In total, this PSO algorithm has 5 parameters (hereafter referred to as constants to distinguish them from the problem solutions),

- ϕ - the single confidence parameter
- N - the number of particles in the swarm
- k - the number of informants for each particle
- *timesteps* - the number of time steps
- *repetitions* - the number of repetitions

Several repetitions are required to get the best average result with few time steps. This is because there is inherent randomness in the PSO algorithm. The results with only one repetitions are too unpredictable, unless the number of time steps is very large. The goal was to find the minimum of Rosenbrock with only 500 time steps.

At the end of the final time step, the final result is calculated from the swarm's g-values. For each repetition, the lowest valued g is found, and the lowest of these is returned as the optimal solution.

2.2 How to use Optimusbeez

To use the Optimusbeez package,

```
>>>import optimusbeez as ob
```

The package has a file ’optimal_constants.txt’ that contains a constants configuration that was found to induce efficient learning. ’fn_info.txt’ contains information about the problem. The default problem is Rosenbrock with a search space of -100 to 100.

Here is a simple example of how to use optimusbeez. First, create an Experiment object.

```
>>>experiment = ob.Experiment()
```

If no arguments are passed to Experiment, it uses the constants configuration and function info from ’optimal_constants.txt’ and ’fn_info.txt’. Then you can run the experiment with the command

```
>>>experiment.run()
```

If no arguments are given to run(), then the experiment will run with the default number of time steps - around 500. A progress bar is shown, and the results will be returned and printed on screen. An animation of the swarm will be shown. To see the animation again, you can do

```
>>>experiment.swarm.animate_swarm()
```

If you wish to run the experiment again, but with a different number of time steps, then you can do

```
>>>experiment.run(10_000)
```

with the required number of time steps given as an argument.

All of the constants can be reassigned as attributes of the Experiment object.

```
>>>experiment.N = 10
>>>experiment.phi = 2.1
```

You can see the current constants configuration and function info with the methods

```
>>>experiment.constants()
>>>experiment.fn_info()
```

A very useful funciton in Optimusbeez is ’evaluate_experiment’. This function runs an experiment several times, plots a histogram of the results, and returns the average result and standard deviation. This function is used to evaluate how well the algorithm performs. ’evaluate_experiment’ takes 3 arguments, the experiment object to be evaluated, the number of allowed evaluations, and the number of times the experiment should be run.

```
>>>ob.evaluate_experiment(experiment, 500, 1000)
```

You can use the Python help() function to get information about what a function does and the parameters and returns of that function.

3 Optimizing the Optimizer

To use this version of PSO, 5 constants are required - ϕ, N, k, time steps, and repetitions. These must be optimized as well, for the algorithm to work properly. This is a true machine learning problem. The only constant that is continuous is ϕ, and the rest are integers. Additionally, each constant has slightly different constraints. ϕ and k are simply constrained by a rectangular search area, similar to x and y in the Rosenbrock function. However, N, time steps and repetitions determine the number of evaluations. The relationship between these is

$$\text{n evaluations} = \text{N} \cdot \text{time steps} \cdot \text{repetitions} + \text{N} \cdot \text{repetitions} \tag{4}$$

If we wish to limit the algorithm to approximately 500 evaluations, then we cannot assign N, time steps, and repetitions randomly at the beginning of the experiment from a rectangular search space.

An attempt was made to use the PSO algorithm to optimize itself. An experiment was created with constants as the particle positions. A special constraints function was defined in order to constrain the positions of the particles.

Because of this, Optimusbeez is now capable of dealing with more than 2 dimensions, integer and non integer parameters, and parameters that have special constraints.

Sadly, using PSO on PSO itself did not work as well as expected. The constants configurations obtained were a good starting point, but manually moving the constants up and down was required to get truly optimal results. The constants optimization only worked well with a number of iterations greater than 300.

3.0.1 Guidelines for choosing constants

- ϕ is most likely in the interval (2, 2.4]. ϕ cannot be less than 2.
- N should be larger for a larger number of evaluations.
- k should be 3 or 4 in most cases.
- repetitions should be at least 3, otherwise randomness affects the final result too much.
- time_steps should be chosen when all other constants are set.

4 Results

The previously introduced function 'evaluate_experiment' was used to test how well the algorithm worked. The constants configuration used was

- $\phi = 2.4$
- N = 9
- k = 3
- repetitions = 3

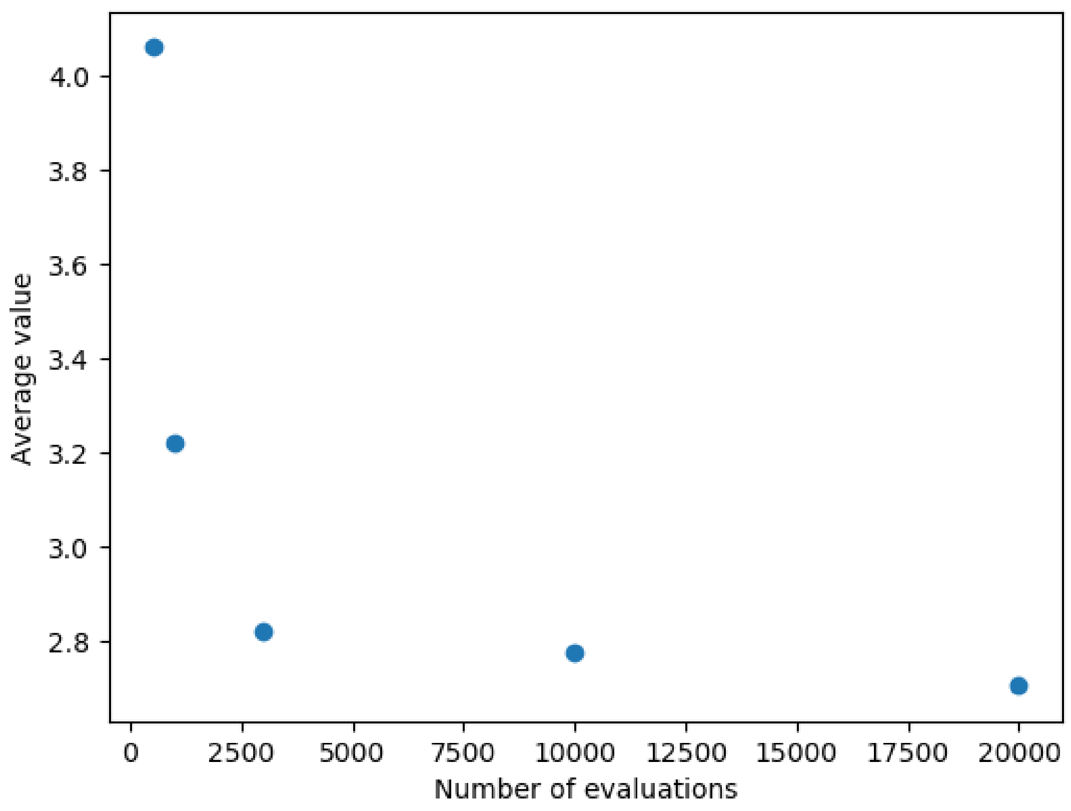

Figure 2: The average value obtained by the experiment over the number of evaluations

Time steps were varied. The results are shown in Table 1 and Figure 2. The average value shows that increasing the number of time steps increased the accuracy of the results, although the constants were optimized for 500 time steps. The results for larger time steps would be even better if the constants configuration was optimized again. If the constants are not optimized, then it does not make sense to use a higher number of evaluations, as the average result plateaus.

For example, for 10 000 evaluations, the optimal configuration seems to be $\phi = 2.62$, $N = 19$, $k = 9$, with 35 time steps and 15 repetitions. This configuration has an average result of 0.0038. This is much lower than the previous result of 2.76. Getting an answer of 1×10^{-6} is not uncommon for this configuration. It is interesting to note that this configuration was obtained only with the constants optimization function without manually adjusting the constants afterwards.

Figure 3 shows a comparison of the histograms for 500 and 100 000 time steps. A different number of evaluation iterations was used, so only the shape of the histogram can be compared. It is clearly seen that the results for 100 000 evaluations are more exact. The swarm finds the global optimum with greater probability with a higher number of evaluations.

Number of evaluations	Average value	Standard deviation
500	4.06	8.0
1000	3.22	7.6
3000	2.82	7.6
10 000	2.76	7.5
20 000	2.71	7.8

Table 1: The results obtained for Rosenbrock with the constants configuration given by 'optimal_constants.txt'.

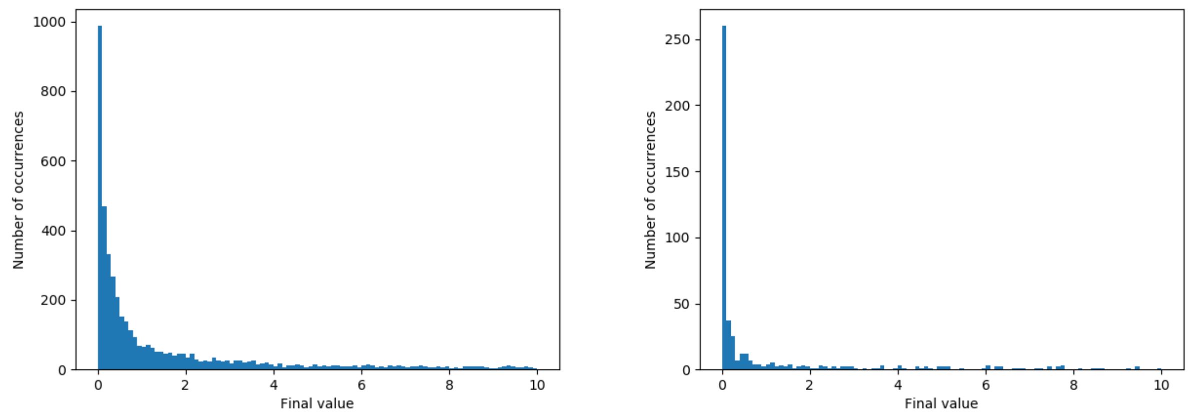

Figure 3: Histograms of 500 evaluations (left) and 100 000 evaluations (right) showing the final value on the x-axis and the number of occurrences of that value on the y-axis, averaged over several runs.

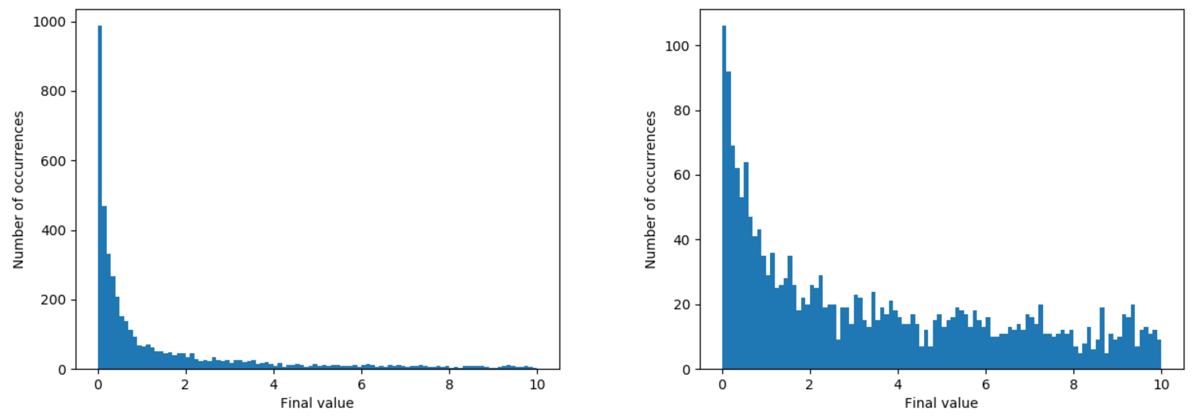

Figure 4: Histograms of final values for Rosenbrock with different parameters a and b, $a = 1$ and $b = 100$ on the left, and $a = 9$ and $b = 75$ on the right.

4.1 Alternative Rosenbrock Definition and Other Functions

The definition of the Rosenbrock function is

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{5}$$

The parameters a and b are usually set to 1 and 100. If these are changed to $a = 9$ and $b = 75$, then the results are entirely different. With the same constants configuration as before, the result is notably worse at 22 ± 26. Figure 4 shows a comparison of the previous results and the results with the new parameters. The worsening results are due to the new shape of the surface. It is harder to escape local minima and the swarm converges prematurely. An attempt was made at optimizing the constants for this new Rosenbrock function but this did not significantly change the results.

Optimusbeez was also tested with the functions Alpine and Griewank. With 500 time steps, the average results are 0.07 ± 0.16 for Alpine and 0.04 ± 0.03 for Griewank. Therefore, Optimusbeez can successfully be used on other functions.

5 How Fast is Optimusbeez?

In terms of run time per evaluation, Particle Swarm Optimization is faster than many other optimization algorithms. But how fast is Optimusbeez? Figure 5 shows a plot of elapsed time against the number of time steps. It is quite satisfying to see a clearly linear increase in time, rather than exponential.

However, in real world problems evaluations may take several hours. If training a neural network takes an hour, then even 500 evaluations are too many. Because of this, the time taken for each iteration is not as important as the required number of evaluations for a satisfactory result. Optimusbeez does not take a long time to run, but it requires at least 500 time steps to consistently get good results.

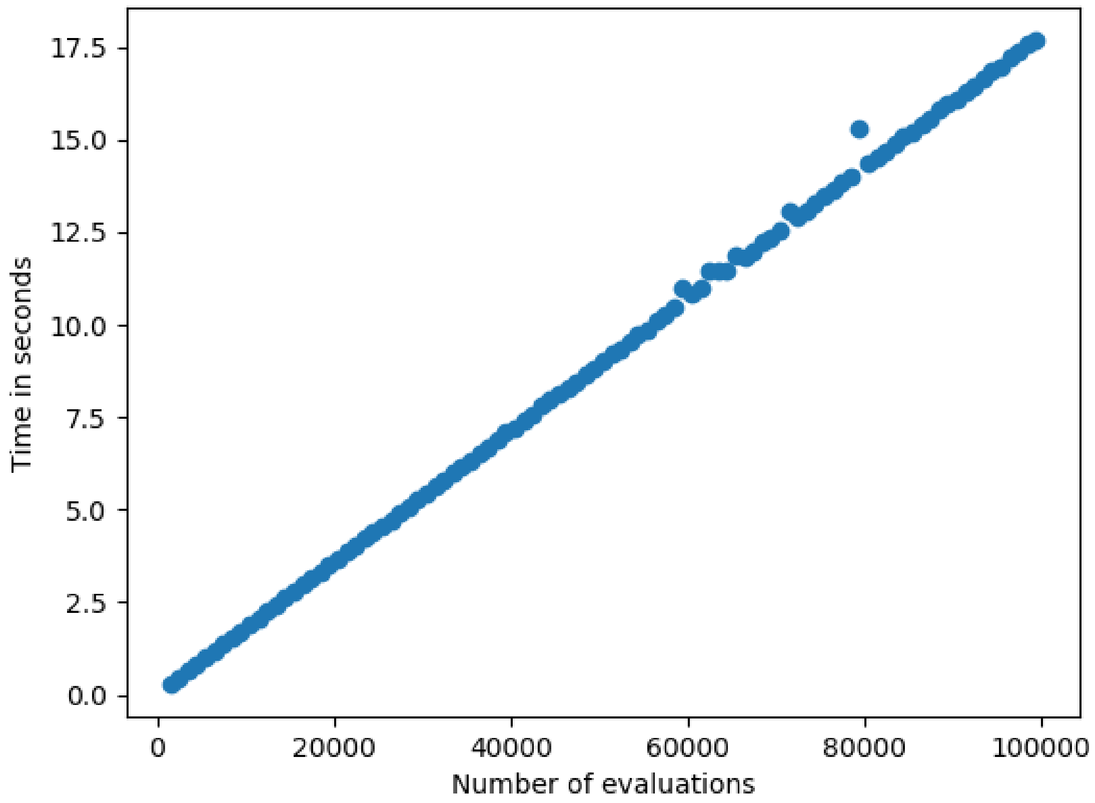

Figure 5: A plot of execution time as a function of the number of time steps

6 Possible Improvements

6.1 Improvements to the Algorithm

Maurice Clerc suggests many improvements to this algorithm in his book. The next logical step would be to decrease c_1 in time to make the swarm converge when it gets closer to the global minimum. With a decreasing c_1, the particles would move less and less with each time step.

Another improvement would be to separate the particles into explorers and memories. This was attempted but the swarm did not converge properly. This was probably because the memories were not implemented correctly.

One difference between this algorithm and PSO(0) from the book is that here each particle chooses k others as informants. The book states that each particle should choose k others to *inform*. The differences in accuracy of the two methods could be minor but this would be an interesting detail to investigate further.

6.2 Improvements to the Package

With some improvements, the package could be very useful in adjusting hyperparameters of other machine learning models. Using the package is relatively simple but another method for setting function info should be created. Giving all that information as a dictionary is complicated and takes too long. The attribute assignment to the Experiment from the fn_info dictionary should at least be done inside a for loop.

More and better unit tests should be written. The usability with higher dimensions is not rigorously tested, and neither is finding integer parameters. These improvements along with other small changes will make this package easily usable by others.

7 Conclusion

I believe Optimusbeez met expectations quite well. Histograms of final values showed that, most of the time, the algorithm found a value less than 1. Considering that Rosenbrock has very steep hills, with values of several thousand not uncommon, Optimusbeez did very well. The package is fun to use and easy to work with. With some improvements, it could be even better.