UNIVERSITAT DE BARCELONA

FUNDAMENTALS OF DATA SCIENCE MASTER'S THESIS

Particle Swarm Optimization (PSO) and two real world applications

Author:
Albert PRAT

Supervisor:
Dr. Gerard GÓMEZ

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Fundamentals of Data Science

in the

Facultat de Matemàtiques i Informàtica

July 1, 2019

UNIVERSITAT DE BARCELONA

Abstract

Facultat de Matemàtiques i Informàtica

MSc

Particle Swarm Optimization (PSO) and two real world applications

by Albert PRAT

Particle Swarm Optimization (PSO) belongs to a powerful family of optimization techniques inspired by the collective behaviour of social animals. This method has shown promising results in a wide range of applications, especially in computer science. Despite this, a great popularity of such method has not been achieved. Since we believe in the potential of PSO, we propose the following scheme to be able to take advantage of its properties. First, an implementation from scratch in C language of the method has been done, as well as an analysis of its parameters and its performance in function minimization. Then, a second more specific part of this thesis is devoted to the adaptation of the method for solving two real-world applications. The first one, in the field of signal analysis, consists of an optimization method for the numerical analysis of Fourier functions, whereas the second, in the field of computer science, comprises the optimization of neural networks weights' for some small architectures.

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance. I would first like to thank my supervisor, Dr. Gerard Gómez, whose expertise in formulating the research topic, setting directives and suggesting methodology was unquestionable.

Secondly, I would also thank Dr. Jordi Vitrià, for the opportunity given to develop a topic of our liking and for the guidance regarding formal issues.

In addition, I thank Núria Valls, my partner in developing this thesis, for her invaluable help provided throughout all this dissertation: from the first moment you have listened to all my doubts and concerns, specially in developing the library and you have spent many hours teaching me C programming language. I have felt understood at every moment and your willingness stands out.

Finally, I would also thank my sister, Dr. Judit Prat, Dr. Àlex Alarcón and Ramon Mir, for their support and patience: you have given me wise advice and emotional support.

Contents

Chapter 1

Introduction

Traditionally, intelligence is thought to be based on individual minds (Marini and Walczak, 2015), not taking into account the relationship between individuals. Nevertheless, we all may agree that enabling this kind of collaboration sometimes can lead to higher performances.

One way in which individuals can be organized is as a swarm. With the word "swarm" we are referring to a set of (generally, mobile) agents that communicate with each other, either directly or indirectly, by acting on their local environment (Jacob et al., 2007). Marini and Walczak, 2015, additionally state that each individual of a swarm is simple, homogeneous and performs elementary tasks. Moreover, its individuals are decentralized and self-organized (Talukder, 2011). Several examples can be found in nature like ant colonies, bird flocks or fish schools.

Moving into the field of computer science, there is a group of Artificial Intelligence methods called Swarm Intelligence (SI) centered around this concept of swarm. Millonas, 1994, proposed five principles for SI in order to determine its behavior:

- **Proximity principle:** the population should be able to carry out simple space and time computations.
- **Quality principle:** the population should be able to respond to quality factors in the environment.
- **Diverse response principle:** the population should not commit its activities along excessively narrow channels.
- **Stability principle:** the population should not change its mode of behavior every time the environment changes.
- **Adaptability principle:** the population must be able to change behavior mode when it's worth the computational price.

One example of SI is the so called Ant Colony Optimization (ACO). Briefly speaking, ACO is a probabilistic optimization technique, aimed to finding the best path along a graph that mimics the wandering behaviour of ants seeking a path between their colony and a source of food (Marini and Walczak, 2015).

Another example of SI is **Particle Swarm Optimization** (PSO). PSO was inspired by the social behavior observed in bird flocks and fish schools. In this case, rather than finding the best path along a graph, PSO is suited to optimize real-valued multidimensional functions. We will see that PSO fulfills the five Millonas' principles.

In this thesis, we will focus on the PS optimization method. Recently, a lot of importance has been given to these non-deterministic methods. Therefore, the aim of this work is to test PSO with several functions to know whether this method is viable for solving more complex problems or not.

For this purpose, a library being able to run the algorithm for the cases we propose has been developed. Apart from that, an extensive analysis of a first approach of the method using simple functions has also been done, as well as a parameter analysis in order to understand both the strong and the weak points of the method. Afterwards, two real-life applications have been done to actually test the degree in which PSO can be generalized for more complex problems.

First, second and third chapters comprise all the tools needed for the development of the library able to run the PSO algorithm. Within these chapters, a theoretical framework is defined as well as details about the implementation of the library. Also in these set of chapters, the parametric analysis of the model is done. Chapter 4 comprises the adaptation of the PSO algorithm for the numeric analysis of Fourier functions and Chapter 5 includes the adaptation of the algorithm for optimizing the weights of a neural network.

Chapter 2

Theoretical framework

Kennedy and Eberhart proposed PSO in 1995. They argued that the main hypothesis which led them to develop PSO is that the members of a fish school can profit from the discoveries and previous experience of all other members during the search of food. This statement was presented by the sociobiologist Edward O. Wilson, in 1975. PSO follows exactly this logic: "the set of candidate solutions to the optimization problem is defined as a swarm of particles which may flow through the parameter space defining trajectories which are driven by their own and neighbor's best performances"(Marini and Walczak, 2015, p. 154).

In other words, swarm particles profit from the previous experience of other particles. Notice that this approach differs from the traditional nature-based methods (like genetic algorithms) in how an improvement to the previous state is made. In PSO, such improvement comes from cooperation and competition among individuals (Marini and Walczak, 2015).

2.1 The basic PSO method

We already defined swarm as a set of organized simple individuals without central control. Let us change the term "individual" for "particle". A potential solution to the optimization problem is the position x of particle i, defined as x_i. Therefore, if we are aiming to optimize D parameters, a potential solution is:

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{iD}), \tag{2.1}$$

and N particles x constitute a swarm, defined as X:

$$X = (x_1, x_2, \ldots, x_N), \tag{2.2}$$

For a moment, imagine a birds flock (X) looking for food. Within the group, each bird looks to a specific direction, which depends on its current position. Later, they communicate among themselves and the bird in the best position is identified. Once determined the "best bird", each bird moves accordingly with a velocity, which depends on the current velocity and some extra information obtained from the swarm. This process is repeated until the desired position is found (Montalvo et al., 2008).

To translate this behaviour into equations we have to go back to 1995, to the first pair of authors which proposed PSO. According to them, each particle position changes following equation 2.3:

$$x_i^{t+1} = x_i^t + \mathbf{v}_i^{t+1}, \tag{2.3}$$

where $\mathbf{v}^{t+1}$ is the updated velocity defined as:

$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \underbrace{2}_{c_1} R_1(p_i - x_i^t) + \underbrace{2}_{c_2} R_2(g - x_i^t). \tag{2.4}$$

In this way, velocity is a vector of length D, which means that for each coordinate we have a different velocity. Since particles are in constant movement, we consider t to be a given moment. Therefore, position x_i^t corresponds to the position of particle i at that specific given moment t. This particle will move with velocity $\mathbf{v}_i^{t+1}$ to reach position x_i^{t+1}. This process, for the N particles define an iteration. Three components help us define velocity at $t+1$:

- $\mathbf{v}_i^t$, which corresponds to velocity at t is also called *inertia* or *momentum* and prevents the particle from drastically changing its direction,
- $2R_1(p_i - x_i^t)$, where R_1 is a randomly generated number from a uniform distribution in the [0,1] interval and p_i is the best position attained by particle i. This part is called *cognitive component* and corresponds to the individual intelligence of the particle. This term increases the probability that the particle returns to their previously best position found, and
- $2R_2(g - x_i^t)$, where g is the global best, that is, the best position attained by the whole swarm of particles at moment t, and R_2 is a randomly generated number similar to R_1. The whole term is called *social component* and identifies the propensity of a particle to move towards the best position (Marini and Walczak, 2015).

Two additional comments should be made to fully understand equation 2.4. The first comment is that R_1 and R_2 are here to ensure that the social component and the cognitive component affect in a stochastic way to the overall change in velocity and the second comment, in contrast, tries to explain why the two constants c_1 and c_2 take value 2.

According to this 1995 paper from Kennedy and Eberhart, $c_1 = c_2 = 2$ because, in this way, it makes the weights for social and cognition parts to be 1, on average, due to the random uniform part. They tried to improve this version of PSO in several ways, however, the most noticeable attempt was the following: they tried to introduce two additional roles: the "explorer" and the "settler". While the role of those particles considered as explorers was to look far away from the target for potential better place, the role of the settlers was to micro-explore regions that were found to be good. In the end, the simpler version seemed to work better.

2.2 Versions of PSO

More than 300 papers related to PSO have been published aiming to improve the basic PSO algorithm. Within this part, we will expose the conclusions of the most relevant studies according to Eberhart and Shi, 2001a, and Xiaohui, Shi, and Eberhart, 2004.

Eberhart and Kennedy, as pioneers of the basic PSO algorithm, proposed a PSO for binary discrete variables with success. This implementation may be used, for instance, for variable selection purposes. Briefly speaking, the position of every particle is binary (either 0 or 1) and is, at each iteration, re-scaled to belong to this interval (since the velocity part keeps being continuous).

One parameter that could affect the performance of the algorithm is the size of the swarm (N). It seems that when N is larger than 50, PSO loses sensitiveness to this parameter and leads to higher probabilities to premature convergence. However, it also increases the exploration ability of the swarm.

Within the whole set of potential improvements of PSO, we will divide them between two groups, depending on which part of the equation is modified. Equation 2.4 can be split in the following two parts:

$$\mathbf{v}_i^{t+1} = \underbrace{\mathbf{v}_i^t}_{\textit{momentum}} + \underbrace{c_1 R_1 (p_i - x_i^t) + c_2 R_2 (g - x_i^t)}_{\textit{individual and social experience}}. \tag{2.5}$$

Therefore, according to equation 2.5, we can perform the following classification:

- Modifications considering *momentum*
- Modifications considering *individual* and *social experience*

2.2.1 Modifications considering *momentum*

The simplest modification you can think of is removing the *inertia* component which means that velocity is memory-less because it is equivalent to assuming that $\mathbf{v}^t = 0$. In other words, particle i stays still until another particle takes over the best global position (Shi and Eberhart, 1998a). Several experiments were done by the former set of authors (Kennedy and Eberhart, 1995) regarding this potential improvement, however, this interpretation was not desirable.

Other proposals were made to improve the initial PSO algorithm, like the one from Shi and Eberhart, 1998a. They introduced a parameter w which they called *inertia weight*. As the name denotes, w accounts for the weight on the *inertia* component. As a consequence, equation 2.4 can be rewritten as:

$$\mathbf{v}_i^{t+1} = w\mathbf{v}_i^t + 2R_1(p_i - x_i^t) + 2R_2(g - x_i^t). \tag{2.6}$$

Their results show, yet, that when w is taken between 0.9 and 1.2 the algorithm performs slightly better in terms of number of iterations and probability to find the global maximum. Figure 2.1 was also retrieved from Shi and Eberhart, 1998a, and is used to support their conclusions. As it can be seen, when w is around 1 the number of failures of the algorithm is kept low. For this reason, practically speaking we consider both equations to be equivalent with the constant optimal w.

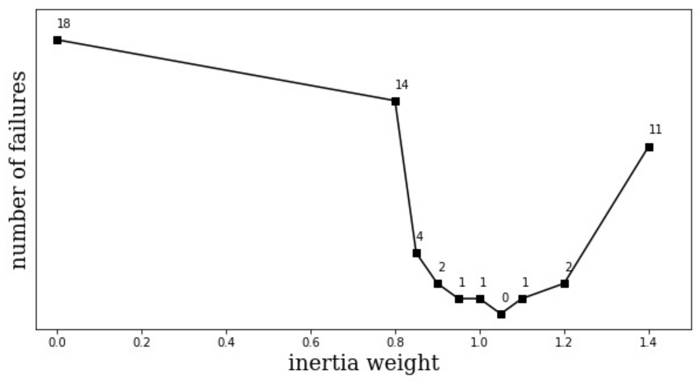

FIGURE 2.1: Number of times PSO did not succeed to find the global maximum as a function of w (Shi and Eberhart, 1998a).

But there was one aspect that actually improved the performance of the algorithm: taking decreasing *inertia weights*. Since it controls the balance between local and global search, it is a way to first give importance to global search and, as cycles pass by, prioritize local search (Montalvo et al., 2008). According to Zhang et al., 2018, a decrease in the value of w is needed for the algorithm in order not to fall in local convergence.

This decrease can be linear or not. Xin, Chen, and Hai, 2009, proposed that w at iteration t should be equal to:

$$\frac{(w_{\max} - w_{\min})(t_{\max} - t)}{t_{\max} + w_{\min}}, \tag{2.7}$$

where $w_{\max}$ and $w_{\min}$ are the initial and final values of w, t corresponds to the actual iteration number and $t_{\max}$ the maximum number of iterations we want our algorithm to perform. Eberhart and Shi, 2001a, suggest that when $w_{\max} = 0.9$ and $w_{\min} = 0.4$, the algorithm performs significantly better.

Until now we have discussed choosing a constant or a linearly decreasing w, yet other authors proposed alternative choices for the *inertia weight*. Apart from both already seen methods, we have a third option, which is defined as stochastic. Within the stochastic choices of w, we have Eberhart and Shi, 2001b and Feng et al., 2007, within others, but any of them seemed to perform better than taking decreasing w.

2.2.2 Modifications considering *individual* and *social experience*

The first modification to consider takes into account the **network topology**. It particularly affects the *social experience* of the swarm individuals. Eberhart and Kennedy, 1995, presented a modified version of the same algorithm where the global best position attained by the whole swarm (g) was substituted by the local best position. Instead of enabling each particle to access the information about the best position attained by the whole swarm, it just could access the best position attained by its neighbors. This number of neighbours was set to 2, according to Bratton and Kennedy, 2007. This approach seemed to lead to lower convergence rates as well as to propensity to be trapped in local minima (Xiaohui, Shi, and Eberhart, 2004).

Another set of parameters that can be tuned are c_1 and c_2, the so called **acceleration constants**. Both parameters help us adjust the importance of each of the experience-related components. As mentioned above, the former recommended values for c_1 and c_2 were 2. However, other approaches have been seen in literature.

While Marini and Walczak, 2015, propose that both c_1 and c_2 should be between 0 and 4, Clerc, 1999, introduces a *constriction method*, which, theoretically, ensures the convergence of the algorithm the same way w does. According to Eberhart and Shi, 2001a, this characteristic is achieved by changing the equation defining $\mathbf{v}^{t+1}$ as follows:

$$\mathbf{v}_i^{t+1} = \mathbf{K}(\mathbf{v}_i^t + c_1 R_1(p_i - x_i^t) + c_2 R_2(g - x_i^t)), \tag{2.8}$$

where

$$\mathbf{K} = \frac{2}{\mid 2 - \phi - \sqrt{\phi^2 - 4\phi} \mid}, \tag{2.9}$$

with $\phi = 4.1 = c_1 + c_2$ we get $\mathbf{K} = 0.729$. Therefore, to assume that the *social component* has the same weight as the *cognitive component* ($c_1 = c_2$) leads to the following equation:

$$\mathbf{v}_i^{t+1} = 0.729\mathbf{v}_i^t + 1.49445 R_1(p_i - x_i^t) + 1.49445 R_2(g - x_i^t). \tag{2.10}$$

This last approach, as we can see, involves modifications on both the *inertia* and *experience*.

2.3 The algorithm

We have seen several approaches for the same optimization method, however, in this part we will explain just the simple PSO algorithm since we consider the same steps can be applied to every modification we have mentioned.

Summarizing, without considering the initialization step, the algorithm works in the following way as long as we are minimizing:

```
for each iteration $t$ do
    for each particle $i$ do
        calculate $v^{t+1}$ according to equation 2.4
        update $x^{t+1}$ according to equation 2.3
        calculate $f(x^{t+1})$
        if $f(x^{t+1}) < f(p)$ then
            update $p$
        end
        if $f(x^{t+1}) < f(g)$ then
            update $g$
        end
    end
end
```

Algorithm 1: Basic PSO algorithm.

This 1st algorithm just considers the iterative part, but we also need the initialization step, which will be discussed in the following section.

2.3.1 Initialization step

According to Marini and Walczak, 2015, there is a general agreement in the literature regarding the initialization for x_i^0, which does not happen for $\mathbf{v}_i^0$:

- x_i **initialization:** Since $x_i \in \mathbb{R}^D$, where D is the number of parameters, x_{ij}^0 will be the initial value of parameter j. This number will follow a uniform distribution in the range of parameter j. In other words, $x_{ij}^0 \sim U(x_{j,\min}, x_{j,\max})$ where $x_{j,\min}$ and $x_{j,\max}$ are the minimum and maximum values parameter j can take respectively.

- $\mathbf{v}_i$ **initialization:** According to Engelbrecht, 2012, the best initialization strategy for $\mathbf{v}_i$ is setting these numbers to 0 or to very small (making them follow a uniform distribution in the range $[-0.1, 0.1]$, for instance). Since the initial position of particles is on the whole domain, using this strategy does not jeopardize the exploration diversity in the initial stage of the algorithm. The other alternative it is proposed in literature is having the same initialization than for x_i. However, Engelbrecht, 2012, showed that, on average, this approach is slower.

Another parameter which has to be initialized and does have an importance on the performance of PSO algorithm is the maximum velocity a particle can travel at ($\mathbf{v}_{\max}$). Without setting a maximum velocity, the algorithm may actually diverge (this phenomenon is the so called *velocity explosion*).

Shi and Eberhart, 1998b, studied how does $\mathbf{v}_{\max}$ affect to the performance of the algorithm. Their results show that if we choose to set a small maximum velocity (i.e. $\mathbf{v}_{\max} \leq 2$), w should be set to 1. On the other hand, if we choose to have greater maximum velocities (i.e. $\mathbf{v}_{\max} \geq 3$), w should be 0.8.

A third case was also proposed: setting $\mathbf{v}_{\max} = x_{\max}$. This method was said to be appropriate in case we lack knowledge about the selection of this velocity. This last option must be followed by setting w also to 0.8.

Nevertheless, we believe that the choice on $\mathbf{v}_{\max}$ must be relative to the problem, therefore, we prefer to stick to the third proposal we have explained regarding this choice or follow the findings of Adewumi and Arasomwan, 2015, which added a parameter μ multiplying both limits of x. Therefore:

$$\begin{aligned} \mathbf{v}_{\max} &= \mu x_{\max}, \\ \mathbf{v}_{\min} &= \mu x_{\min}, \end{aligned} \tag{2.11}$$

where $\mu \in (0, 1]$, for each parameter j.

By explicitly setting a limit on $\mathbf{v}_{\max}$, *velocity explosion* is controlled. However, there are, at least, two more ways one can avoid divergence. Both methods have already been explained and are:

- Adding decreasing *inertia weights*.

- Adding the constriction factor $\mathbf{K}$.

Chapter 3

Implementation

3.1 Experimental setup

Before starting with the implementation, we must specify the experimental characteristics. All the code generated in this thesis implementation is done in C language (Ritchie, 1993) because it is a compiled language and so has a fast performance when implementing iterative processes with simple calculations. Also codes in C can be easily transformed into transparent libraries for users.

The development environment that we have chosen is CLion (JetBrains, 2019), which is a cross-platform IDE for C and C++ that has a compact integration of a terminal, which is useful for non Linux users.

Apart form the standard C libraries there has been one extra library included in this project thesis:

- FFTW library (Frigo and Johnson, 2005): is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data.

This library has been used in this thesis in the first application of the PSO algorithm for DFT computations further explained in the respective section.

3.2 Software design

As a basic implementation of the PSO algorithm in this thesis we have started by creating a project in which one can specify any finite 3-dimensional function to minimize together some basic configurations so that its execution converges to one point of the function, which will be the minimum. Consequently, we have decided that the architecture of the mentioned program should be structured in a modular way such that it can be further used as a public library to minimize any finite three-dimensional function.

More in detail, the base program consists of two different modules named `logica` and `utils`, the first one contains all the functions needed to execute the particle swarm optimization procedure. The second one contains more general use functions that are not related with the inner functions of the algorithm. But the way in which all the functions of the modules must be used in order to execute the algorithm is shown in the `main` file, which include all the modules said before.

There are only two things that a user must create in order to change the execution of the algorithm:

- The **configuration file** (`config_file.txt`): This is a text file that contains the basic configuration parameters for the program to run one single optimization procedure. It has been done as a text file and not as constant declarations in the code in order to abstract them from the inner functioning of the algorithm as the configurations are treated as generic. As a consequence, the name of this file needs to be passed as an argument when running the program.

The default configuration file that is given in the program has the following parameters explained below:

1. Population size of the swarm: specifies the number of particles that the swarm will have moving through the parameter space in an integer format.
2. Number of parameters to optimize: specifies the number of parameters that each particle will have in an integer format
3. Range of the parameters: specifies the minimum and the maximum value that a single parameter can have in [float , float] format. There are d range lines in the configuration file, where d is the number of parameters, so that there is a specific range for every parameter.
4. Maximum velocity fraction: specifies the maximum velocity that a parameter can have, which is computed with the equation 3.1.

   With this variable we can regulate the maximum velocity of each parameter as a fraction of its range space in order to avoid the exploding velocity problem.

- The **function** to minimize (f): This is a function located at the `main.c` file that must have two input float parameters and return a float value, which will be minimized by the algorithm.

$$max\_v = max\_v\_fraction * (range\_max + |range\_min|). \tag{3.1}$$

Once specified the above parameters, the project is ready to be executed until it finds a convergence point as the minimum of the minimum fit value that will be achieved when the minimum value returned by the function f.

3.3 Parametric study

Our algorithm supports four different PSO methods:

0. Basic PSO without limiting maximum velocity at which particles can fly.
1. Basic PSO with a fixed velocity limit (Kennedy and Eberhart, 1995).
2. PSO with decreasing *inertia weights* (Xin, Chen, and Hai, 2009).
3. PSO with constriction factor **K** (Eberhart and Shi, 2001a).

All methods have been extensively explained in Chapter 2. In this section, we will provide some evidence to explain how does the algorithm performs in each of these four methods for the Ackley's function, which serves as a benchmark for optimization problems.

The aim of this study is to choose the most appropriate method and parameters for the practical applications in this thesis. In all methods above, we consider the algorithm to have converged if the best fit value (g) does not change in a period of 100 iterations.

As said at the end of the previous chapter, we may have divergence if particles fly too fast. While methods 2 and 3 already incorporate mechanisms to avoid *velocity explosion*, method number 1 has to be provided with some $\mathbf{v}_{\max}$, the maximum velocity at which particles can travel as a fraction of the range explained above. In Method 0 though, velocity is not limited at all. For this reason, this method was rapidly discarded because of frequent divergence.

Let's discuss the first method. As said, according to our notation we have two decisions to make: N and $\mathbf{v}_{\max}$. Figure 3.1 shows the behaviour of the optimization method provided both parameters. The size of the circle represents the precision of the procedure measured as the fit value, and darker colors mean higher Ns. Iterations and fit values are computed as the average of a thousand of runs of the algorithm for each pair of ($\mathbf{v}_{\max}$, N).

As we can see, higher values of $\mathbf{v}_{\max}$ lead to worse precision and the other way around. This is due to the fact that particles can fly much faster with higher limits of velocities, therefore, when particles are closer to the minimum, they are not able to explore the region precisely. In other words, the probability that a particle overflies the minimum is higher with higher values of $\mathbf{v}_{\max}$. On the other hand, increasing $\mathbf{v}_{\max}$ also leads to less iterations, on average.

We can observe a similar behaviour looking at the size of the swarm. This parameter controls the exploration capabilities of the swarm. It can also be seen that if the swarm is formed by less than five particles, the algorithm is less precise. Generally speaking, computational cost increases as N grows due to the fact that a fit value has to be calculated for each particle of the swarm. As a consequence, looking at the figure we can see, though, that a population size of 20 particles is enough to not limit exploration capabilities of swarm without barely losing precision.

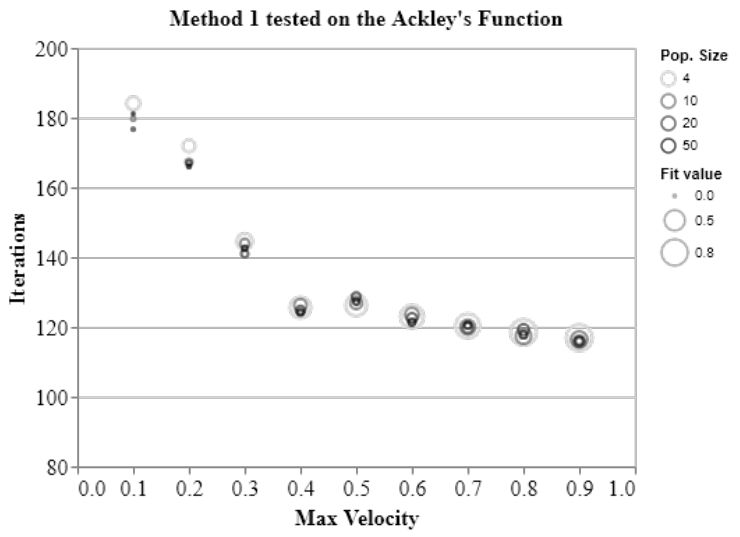

FIGURE 3.1: Behaviour of method 1 on the Ackley's function.

As a conclusion for the first method, we can say that the most balanced parameters are $N = 20$ and $\mathbf{v}_{\max} = 0.4$. However, we can always play with these parameters knowing the effect the switch will produce. Besides, we can compare the first method with these parameters with the other additional methods we propose to determine which method would fit better each of the two applications we are proposing in this thesis. As explained above, in these additional two methods we do not need to choose any maximum velocity. Therefore, the only parameter we have to fix is N, which will be set to 20 because of the reasons also explained above.

The comparison of the convergence plot for each of the methods can bee seen in Figure 3.2. Since we are running a thousand times each of the methods, the fit value plotted is the average of the whole set of runs at each iteration. In the same way, these vertical bars show at which point half of this thousand of runs had already converged. In other words, if we look at the second method for a moment, we can say that at iteration 105 approximately, 500 out of the 1000 runs of the algorithm had already meet the convergence criteria. Using this logic, it can be clearly seen that the best method, by far, is the second one, in which as time goes by, the parameter we called *inertia weight* decreases. It increases the local exploration of potential regions where extrema can be found.

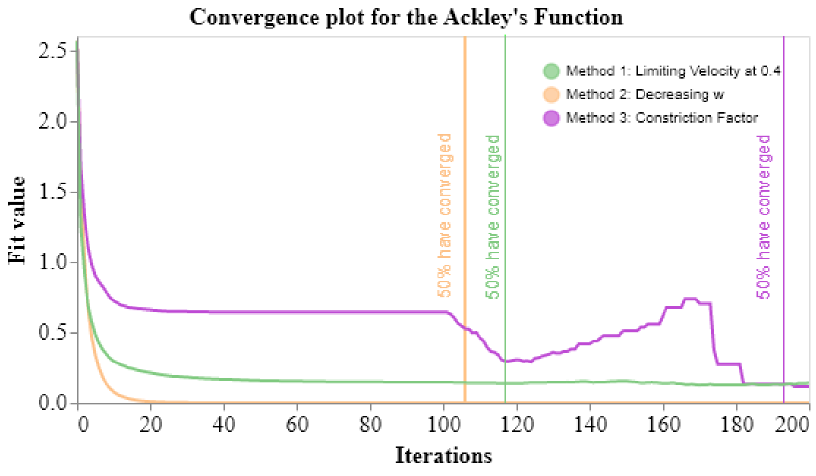

FIGURE 3.2: Convergence plot for each of the 3 methods we proposed.

Another aspect worth mentioning is that method 1 initially converges faster. However, we are missing precision when we are close to the extrema, which does not happen for the second method. These results are coherent with the findings in Eberhart and Shi, 2001a, Montalvo et al., 2008 and Zhang et al., 2018. On the other hand, the constriction factor method works worse as expected. Clerc, 1999 claimed that this method ensured convergence the same way w did, which, in our results, did not happen.

As a conclusion, we can adjust the parameters of the algorithm depending on our needs. If our goal is just to have a high precision, method 2 works best. Not only converges faster, but it achieves a higher precision. Besides, we could always play with different $w_{\max}$ and $w_{\min}$ for this same method depending on our needs. However, if we consider we need a faster convergence in the first iterations method 1 would fit well enough.

3.4 Tests and results

After studying the parameter dependency of the algorithm in one function, we will evaluate the same algorithm with several other functions with different characteristics in order to test how the algorithm performs in this cases. We assume that to achieve the best performance for every function, a complete parametric study should be made, just like in the previous subsection, but in this case we only want to have an idea of how the algorithm is able to achieve good results with simple functions and more complex ones.

3.4.1 2D Parabola

Even though this is a very trivial example, it lets us have a clear idea of the behavior of the algorithm. We can analyze the precision obtained with the convergence parameters and its performance on the number of iterations that took it to converge.

The parabola function used in this case is defined by:

$$f(x,y) = x^2 + y^2. \tag{3.2}$$

In figure 3.3 we can see the contour plot of the function that we want to optimize, in other words, find its minimum value. The outputs resulting of the convergence of one thousand executions of the optimization of the mentioned function are scattered in it with the color according to the density of the points, meaning the more yellow the more density and blue the other way around.

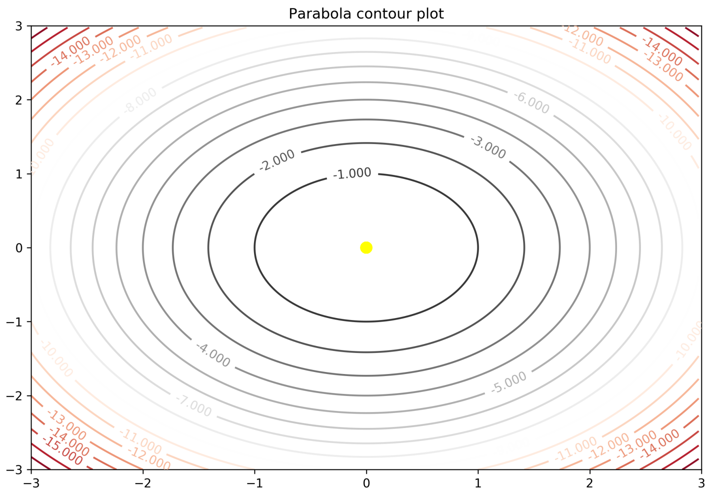

FIGURE 3.3: Contour plot of the parabola function with the convergence points of 1000 executions of the algorithm.

With the results obtained, it can be seen that clearly all the executions have converged to the same point that also matches with the minimum value of the function situated at $[0,0]$.

3.4.2 Ackley's function

In this case we want to see how the algorithm resolves a more complex function typically used to check optimization performance. The Ackley's function is nonconvex function proposed by Ackley, 1987, defined as:

$$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5\cos 2\pi x + \cos 2\pi y} + 20 + e. \tag{3.3}$$

It we repeat the same experiment as above and plot the results in figure 3.4, we can see that this time not all the convergence points have reached the true global minimum of the function, situated at $[0, 0]$, some of them, in a much lower density have fallen in local minima points near the global one. Although most of them do have found the right global minima.

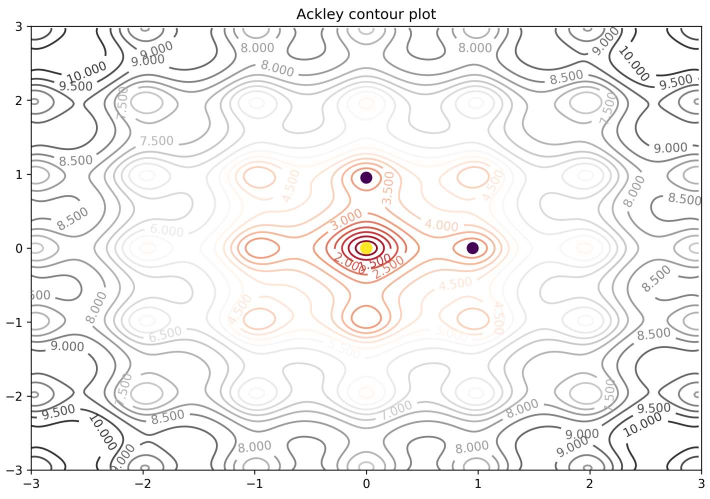

FIGURE 3.4: Contour plot of the Ackley's function with the convergence points of 1000 executions of the algorithm.

3.4.3 Multiple global minima function

Although we have checked the performance of the algorithm for functions with one global minima and several local minima, we still want to know how it behaves when it has more than one global minima to find. In this case we will use a function that is defined in the range $x \in [-2, 2], \quad y \in [-1, 1]$, and has two global minima at $[-0.089842, 0.712656]$ and $[0.089842, -0.712656]$.

This function is defined by:

$$f(x, y) = x^2(4 - 2.1x^2 + \frac{1}{3}x^4) + xy + y^2(-4 + 4y^2)). \tag{3.4}$$

From the results in the figure 3.5 it can be seen that the convergence points are equally distributed among the two global minima. As there is only one output point from the algorithm, we can only find one global minima at a time, but the results are consistent with the two points so we can conclude that the algorithm performs well given this multiple solution situation.

3.4.4 Rosenbrock's function

Finally we would like to test the algorithms performance in a more non-trivial function, like the Rosenbrock's function (Rosenbrock, 1960). This is a non-convex function commonly used as a performance test problem for optimization algorithms. The main characteristic of this function is that it has a global minima hidden inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial. To converge to the global minimum, however, is difficult.

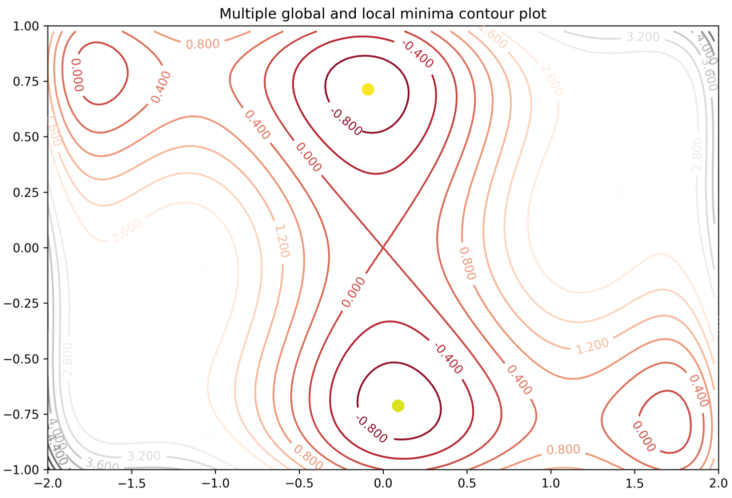

FIGURE 3.5: Contour plot of the multiple global minima function with the convergence points of 1000 executions of the algorithm.

The function is defined by:

$$f(x,y) = (a-x)^2 + b(y-x^2)^2. \tag{3.5}$$

In this case study we will use the values $a = 1$ and $b = 100$ which lead to a global minima at $[1,1]$.

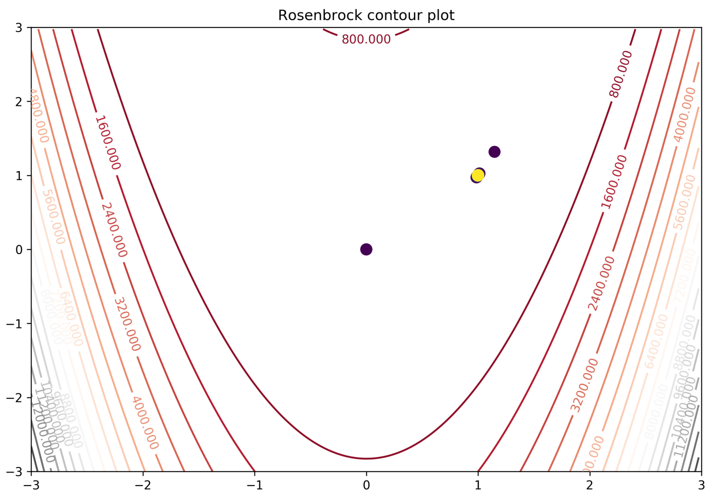

FIGURE 3.6: Contour plot of the Rosenbrock's function with the convergence points of 1000 executions of the algorithm.

Analyzing the results in figure 3.6 we find that all the points have fallen inside the valley but some of them have not been able to find the global minima point, although the greatest part of them do have found the global minima, marked as the yellow point.

3.4.5 Comparison

Once exposed the precision results obtained from the execution of the PSO algorithm among four different functions, one shall not ignore the number of iterations taken to achieve those results in each case.

In order to do so, a box plot is exposed in figure 3.7, where one can see the distribution of the number of iterations for the one thousand executions in each of the four functions.

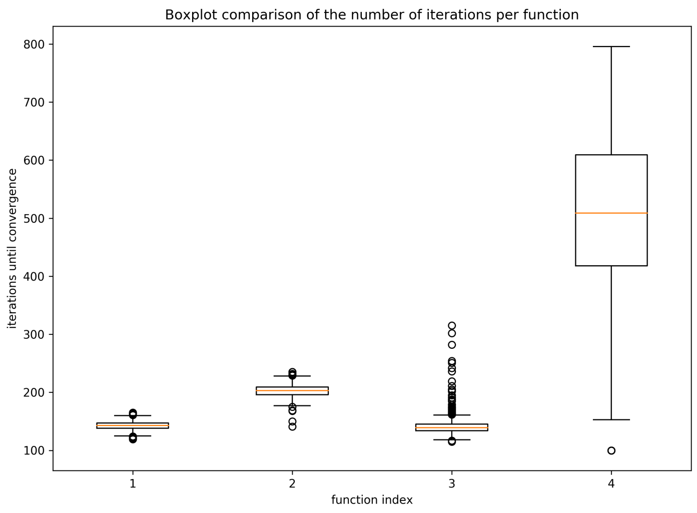

FIGURE 3.7: Box plot comparison of the number of iterations for each function. From left to right: parabola, Ackley, multiple global minima, Rosenbrock.

The most remarkable difference id the higher mean number of iterations from the Rosenbrock's function. As mentioned in the respective section, it is the most difficult function to optimize among the four chosen, this is clearly supported by the results in iterations although it has a large variance. It is worth to mention that still it got good accuracy in the convergence points.

Talking about the other functions, the most robust function in convergence iterations is the parabolic one as expected. It can also be seen that with the two minima function, the mean number of iterations is lower than with the Ackely's function but the first one has a higher variance. This makes sense having in mind that the double global minima could be confusing for the particles of the swarm.

From all the results analyzed in this chapter, we can conclude that the implemented version of the PSO algorithm in this thesis has a good behavior and so we can proceed to test it in more complex problems from real world situations.

Chapter 4

Numerical Fourier analysis

Entering the field of signal analysis, the determination of frequencies and amplitudes of quasi-periodic functions is a complex problem that has been solved in a numerical way. However, this procedure requires a considerable amount of computational time due to equation solving.

A different approach to solve the problem is proposed in this chapter, where the frequencies and amplitudes of a multi-frequency signal are detected, starting from equally-spaced samples of if on a finite time interval, using the particle swarm optimization technique.

4.1 Brief introduction to signal processing

In the field of signal analysis every waveform function that has a certain frequency and amplitude can be expressed as a function of time. However, this functions can be represented or approximated by sums of simpler trigonometric functions. This decomposition procedure can be done with the Fourier transform (FT) technique, that decomposes a function into its primary oscillatory components which are determined by frequencies and its corresponding amplitudes. The method that applies the FT to an input signal is called discrete Fourier transform (DFT) and gives as an output a complex-valued function of frequency.

4.2 The frequency analysis problem

Among all the possible combinations of signals in a certain frequency space, we will take just one example for this thesis application, which can be expressed as follows.

Given N samples $\{f(jT/N)\}_{j=0}^{N-1}$ of a real-valued function $f(t)$, equally spaced on the interval $[0,T]$, the main objective is to determine the trigonometric polynomial,

$$Q_f(t) = A_0^c + \sum_{l=1}^{N_f} (A_l^c \cos 2\pi\nu_l t/T) + (A_l^s \sin 2\pi\nu_l t/T), \tag{4.1}$$

whose frequencies $\{\nu_l\}_{l=1}^{N_f}$, and amplitudes $\{A_l^c\}_{l=0}^{N_f}$, $\{A_l^s\}_{l=1}^{N_f}$, are good approximations of the ones of $f(t)$. The number of frequencies, N_f, has to be determined in terms of an input parameter.

A classical approach to find the above polynomial is based on looking at the peaks of the modulus of the DFT of the signal. Each peak of the function is taken as an approximation of a frequency of the signal, whose amplitude is also approximated by the amplitude of the corresponding peak. The results obtained this way will only be approximations as by definition there will be an error of order $1/T$, where T is the length of the interval of the samples in the frequencies and amplitudes detected, due to the discretization of the signal.

Because of that, many improvement methods have been proposed with excellent results, such as Laskar, 1999 and Gomez, Mondelo, and Simó, 2010. But both methods require complex computations that may be time consuming. Consequently, in this chapter, the basic PSO algorithm will be adapted to solve the frequency identification problem of the DFT modulus of a given signal.

4.3 Adaptation of the algorithm

The multi-frequency function that has been used as the input signal to analyze follows the equation:

$$f(t) = 0.5e^{i2\pi\nu_1 t} + e^{i2\pi\nu_2 t} + 0.4e^{i2\pi\nu_3 4t}, \tag{4.2}$$

which consists of the sum of three primary tones supported at frequencies $\nu_1 = 0.1$, $\nu_2 = 0.3$ and $\nu_3 = 0.4$.

4.3.1 First approach: all at once

Following the structure of the basic PSO implementation explained in section 3.2 there are two different aspects of the low level performance that have been changed in this approach:

1. The **particle parameters**: As there are only six unknown parameters in the signal, three frequencies and three amplitudes, each particle of the swarm will represent a three-frequency signal and will have six parameters.

2. The **fit function**: The chosen formula to calculate the fit value given a particle is summarized in the following equation.

$$fit = \sum_{j=0}^{N-1} \left| f\left(\frac{jT}{N}\right) - \hat{f}\left(\frac{jT}{N}\right) \right|. \tag{4.3}$$

   where $f(w)$ stands for the DFT modulus of the objective function and $\hat{f}(w)$ stands for the DFT modulus of the signal formed with the parameters of a certain particle p.

Tests and results

The results obtained from the analysis of one thousand executions of the algorithm over the previously explained function are summarized in the table 4.1.

	Mean (μ)	Standard deviation (σ)
Fit value	2.8108	0.8933
Iterations	14206	4919
Time per iteration [s]	5.1762×10^{-4}	3.9178×10^{-5}

TABLE 4.1: Summary of the results obtained with the first approach implementation.

As it can be seen, the mean value for the fit is pretty high although it takes a lot of iterations to converge. This leads us to conclude that the algorithm gets lost on the parameter space when optimizing all the six parameters at once, because it is not able to keep searching for better solutions as it gets stuck in local minimums. For this reason, there is still a need to improve the convergence values achieved.

4.3.2 Second approach: peak by peak

The second proposed approach is based in the classical method that identifies each frequency component looking for the highest peaks in amplitude of the modulus one by one.

This procedure implies the segmentation of the original problem into thee different subsections, each of them with the objective of finding a different peak of the modulus function. It has been implemented using three different optimization steps sequentially executed where at each step the objective function is updated by the subtraction of the previous step DFT modulus on the original objective modulus.

Regarding low level configurations only changes in the particle parameter have been done. In this case only two parameter particles are needed.

Tests and results

The results obtained from the analysis of one thousand executions of the algorithm using this second approach are summarized in table 4.2 segmented by steps.

		Mean (μ)	Standard deviation (σ)
Fit value	First step	4.7167	1.4775
	Second step	2.5654	0.5912
	Third step	1.3617	0.8541
Iterations	First step	10559	1213
	Second step	11033	2616
	Third step	10839	1505
Time per iteration [s]	First step	3.4086×10^{-4}	3.0439×10^{-5}
	Second step	3.4100×10^{-4}	3.0416×10^{-5}
	Third step	3.4112×10^{-4}	2.9891×10^{-5}

TABLE 4.2: Summary of the results obtained with the second approach implementation.

From the above results one can make three conclusions. First of all, when looking at the mean fit values for the three steps, one can see that the fit value of the third one, at the moment when all the three peaks have been detected, has decreased more than a half compared with the fit value from the first approach. Also, the number of iterations per step has decreased but taking into account that the three steps are executed sequentially, the total number of iterations per execution will be the sum of the three, leading to a much higher number of iteration than the first one. Regarding the computational time per iteration, it is maintained through the steps and has decreased compared with the first approach. This can be explained as the complexity of the calculus required has been reduced in the second approach although they are executed much more times.

Chapter 5

Neural Network weights optimization

Another application of PSO can be found in the Deep Learning setting, more concretely, in the optimization of Neural Networks weights. To be able to optimize these weights, we have to be knowledgeable about the role these weights play. For this reason, we will introduce briefly the learning problem, how a Neural Network is structured as well how to train a network with such characteristics.

5.1 The machine learning problem

In general, learning can be defined as the modification of behavior tendency according to experiences which have been acquired. While a human starts learning from the day they are born and never stops doing it, the same for a machine is hard to imagine at this moment. Nevertheless, methods with the aim of emulating, at least partly, human learning do exist.

According to Domingos, 2012, machine learning algorithms can figure out how to perform tasks by generalizing from examples. To do so, therefore, we need examples (data) and a task to perform.

We could think of data as N vectors of length n. Therefore, while a sample can be defined as $x_i = (x_{i1}, x_{i2}, \dots, x_{in})$, a stack of samples constitute our data. In this way, data can be arranged in a matrix $X \in \mathbb{R}^{N \times n}$. In particular cases, a sample may have an additional dimension, y, which is the label of the sample. If this mapping from x to y exist for all i from 1 to N, the problem is said to be supervised.

For a machine learning algorithm to learn in this setting, we have to expose them to some samples. This set of samples $\mathcal{D} : \{(x_i, y_i)\}_{i=1}^{N}$ is called the training set. In this way, the machine learning algorithm actually learns how to map each sample in the training set to its label. However, more than knowing how to perform such mapping, we are interested in mapping a sample which has not been seen for the algorithm. If we are able to do such thing, we would be attaining generalization, which is our goal. The set of samples unseen by the algorithm that will allow us to test whether the algorithm has learned to generalize for the task we are aiming to perform is called "test set". The machine learning pipeline can be visualized in Figure 5.1.

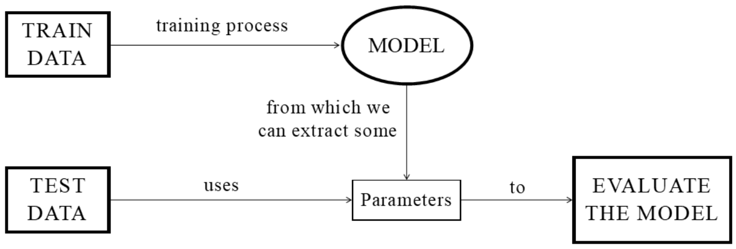

FIGURE 5.1: Representation of the machine learning pipeline.

The other ingredient we need to design a machine learning algorithm is the task we are aiming to perform. Depending on the nature of this task, we can have classification or regression problems. Given a discrete set of m different classes, i.e. m different labels, a classification task consist of being able to tell which class does a sample belong to. On the other hand, in a regression problem we are aiming to predict a real valued quantity, often in a continuous space.

These tasks, that are sometimes effortless for human beings, proved to be immensely difficult. From early 19th century, several methods to attain such achievement have been proposed. Bayes' theorem, Least Squares or Markov chains belong to this group of methods. However, for the purpose of this thesis we are interested in literature considering Artificial Neural Networks. Frank Rosenblatt, in 1957, published "The perceptron", which is considered to be the first paper regarding this topic. From then on, researches have studied deeply neural networks.

5.1.1 Definition of an Artificial Neural Network

According to Kriesel, 2007, a neural network is formed by two sets, N and V, and a function $w : V \to \mathbb{R}$. N is a set of neurons, V is a set $\{(i,j) \mid i,j \in \mathbb{N}\}$, which define directed connections between neurons and $w_{i,j}$ is defined as the weight of connection between neurons i and j. These weights can be arranged in a square weight matrix $W \in \mathbb{R}^{N\times N}$. The row number and the column number indicate where the connection begins and ends respectively. If the connection between neuron i and neuron j is nonexistent, the position i,j of the matrix is set to the numeric 0.

Moreover, to the end of emulating the behavior of the human brain, each neuron can be either activated or not. The process of activation can be visualized in Figure 5.2. It has three different steps and each step is performed by a function:

1. **Propagation function**: since neuron j often receives several values from other neurons, the propagation function transforms the vector of inputs to a scalar, which we will call net_j. It is normally computed as a weighted sum of outputs from other neurons, i.e. $\text{net}_j = \sum_{i\in I}(o_i \cdot w_{ij})$ where I is the set of neurons connected to j and o_i is the output of the i^{th} neuron belonging to I.

2. **Activation function**, a_j: the output of the propagation function, net_j is passed to the activation function. If this exceeds a threshold Θ, the neuron is activated. Formally, $a_j = f_{\text{act}}(\text{net}_j, \Theta_j)$.

3. **Output function**, f_{out}: calculates the values which are transferred to other neurons connected to j (Kriesel, 2007, p. 38). It is often the identity function which makes this translation, therefore, $f_{\text{out}}(a_j) = o_j = a_j$.

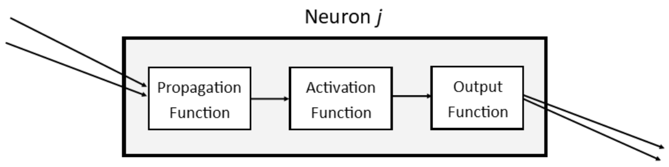

FIGURE 5.2: Activation process of a single neuron j.

After being knowledgeable of the core components of a neural network, we can think of a practically infinite number of possible designs. One of the simplest architectures we can think of is the so called **feedforward** neural network design.

5.1.2 Feedforward Neural Networks

In such networks, neurons are grouped in three different types of layers: one input layer, n_h hidden layers and one output layer. Moreover, if we consider these groups to be stacked sequentially, we can number each layer l with an integer from 1 to L, the total number of layers. Another characteristic of forward neural networks is that connections of a neuron in one layer are only set in a directed way to neurons in the next layer. More particularly, if neuron i is connected to every neuron in the next layer, the network is considered to be completely linked.

For clarification purposes, consider the network in Figure 5.3. This network is a simple completely linked feedforward neural network with 6 neurons, grouped in three different sets, the input layer, with neurons i_1 and i_2, a hidden layer of 3 neurons (h_1, h_2, h_3) and one output neuron (Ω):

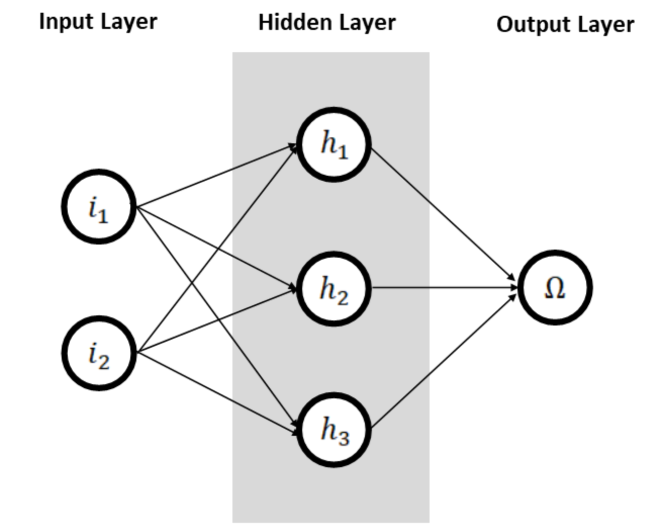

FIGURE 5.3: A simple neural network architecture.

Therefore, our weights matrix W for this architecture has non-null elements over the diagonal in the following way:

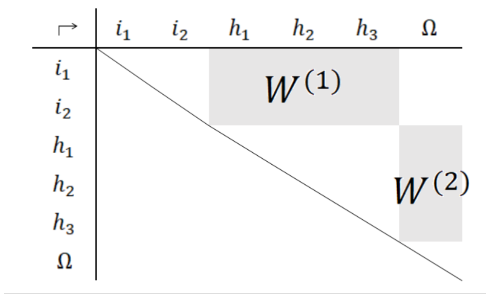

FIGURE 5.4: W matrix for the example we are working with.

Notice that we have split this matrix W into two different ones. $W^{(1)}$ and $W^{(2)}$ are a subset of the total set of weights. The former matrix gathers all weights connecting neurons i_1 and i_2 to neurons h_1, h_2 and h_3. The latter matrix, on the other hand, relates these three neurons to the output neuron Ω. For general forward architectures, $W^{(l)}$ is the subset of weights relating layers l and $l+1$.

5.1.3 Feedforward neural networks and data

Finally, we must look at the fact that many types of neural networks permit the input of data. A feedforward neural network is not different. As defined above, these data are processed by the network and produces an output (Kriesel, 2007). This process of going from the input layer to the output layer will be explained in this section.

Let us, for example, look at the example above. Allocating 2 neurons to the input layer implies that the length of a sample, n as defined above, is 2. Thereby, having N samples implies that the network will receive N instances of two numerical inputs. Similar occurs for the output layer. Allocating 1 neuron to this layer means that the network will just output one numerical value. For a general case, the output can be m-dimensional and arranged in vector $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$. The length of this vector will be the amount of neurons we decide to put in our output layer of the network. But we are missing the key question: how does a feedforward neural network process these data X? In other words, how do we go from the input data to the output?

To make such forward pass, we need to relate how a neuron is activated (see Section 5.1.1) with the fact that we have layers with more than one neuron. Recall that the propagation function commonly used to compute the net input for neuron j is $\sum_{i \in I}(o_i \cdot w_{ij})$. With this notation, if we are looking at the first neuron of the first hidden layer for our concrete example with two numerical inputs, we would compute the whole activation of this neuron as:

$$\begin{aligned} \text{net}_1 &= o_1 w_{11} + o_2 w_{21}, \\ a_1 &= f_{\text{act}}(\text{net}_1), \\ o_1 &= a_1. \end{aligned} \tag{5.1}$$

Notice we are using w_{11} and w_{21}, which can be found in positions $(1,1)$ and $(2,2)$ of matrix $W^{(1)}$. Besides, there exist a bunch of activation functions we can use, however, we will present both *ReLu* and *sigmoid* activation functions. For $j = 1$:

$$f_{\text{act}}^{ReLu} = Max(0, \text{net}_1), \quad f_{\text{act}}^{sigmoid} = \frac{1}{1 + e^{-\text{net}_1}}, \tag{5.2}$$

In a general way, we can proceed using equation 5.1 for each of the neurons in layer l. Moreover, the activation function is often decided for a whole layer. Therefore, we can arrange the both net inputs and activated values in matrices. The former matrix will be called $Z^{(l+1)}$. In our example:

$$Z^{(2)} = \begin{pmatrix} \text{net}_1 & \text{net}_2 & \text{net}_3 \end{pmatrix}. \tag{5.3}$$

Afterwards, we can apply the activation function directly to the net input for each of the neurons. $a^{(l+1)}$ will be, then, the activation values arranged in a matrix with the same dimensions as $Z^{(l+1)}$ for the whole set of neurons of layer l. Also in our example:

$$a^{(2)} = \begin{pmatrix} f_{\text{act}}(\text{net}_2) & f_{\text{act}}(\text{net}_2) & f_{\text{act}}(\text{net}_3) \end{pmatrix}. \tag{5.4}$$

Therefore, we can process our data with just 2 steps per layer: a matrix multiplication and an element-wise application of the activation function. To clarify, the whole process of going from the input to the output can be written as:

$$\begin{aligned} Z^{(2)} &= XW^{(1)}, \\ a^{(2)} &= f_{\text{act}}(Z^{(2)}), \\ Z^{(3)} &= a^{(2)}W^{(2)}, \\ a^{(3)} &= f_{\text{act}}(Z^{(3)}), \\ &\vdots \\ Z^{(l+1)} &= a^{(l)}W^{(l)}, \\ \hat{y} &= f_{\text{act}}(Z^{(l+1)}). \end{aligned} \tag{5.5}$$

This process can be repeated as many times as needed to teach a neural network how to do a certain task, as long as the conditions of learning hold.

5.2 Training a feedforward Neural Network using PSO

Due to the fact that X is fixed and we can't modify it, looking at equation 5.5 we can see that by adjusting W, the neural network can yield one prediction or another. Hence, the learning process consists of learning the optimal set of weights that yield the most accurate predictions. If the network is able to give accurate predictions for unseen samples (test samples as explained in section 5.1), we can say that the network has learned to perform either the classification or the regression task.

As explained above, we need to expose the network to a set of samples to allow the algorithm to learn from them. As you can see in Figure 5.1, this learning process is undertaken just using training samples. Entering into detail, we perform a forward pass of these training samples and obtain a prediction ($\hat{y}$).

Once we have our prediction, we need to compare it to the actual label (y). Here is where the learning process happens. Since at the end of the day we want our predictions to be similar to y, our aim is to minimize this distance. To measure such distance, we can use any function we want, depending on the problem we are aiming to solve. This function is the so called "loss function" and is our objective function in the optimization process. Until now, the best optimizer for neural networks in most situations seems to be the gradient descent or its variants. One of its problems, though, is the possibility that gradients may explode or vanish. One way to solve this problem is correctly initializing these weights.

In our case, though, we will not be using gradient descent but Particle Swarm Optimization algorithm to adjust these weights. Each particle of the swarm will be the whole set of weights needed to compute $\hat{y}$. In this way, we get rid of gradient-related problems.

To summarize, from a simplified perspective, each iteration of the process of training a neural network using forward propagation and PSO has the following steps (without the initialization step):

1. Adjust the weights using PSO.
2. Compute $\hat{y}$ using these weights and according to equation 5.5.
3. Assess the value of the loss function and proceed again. If the process is designed correctly, this loss should be smaller than the loss of the previous iteration.

Eventually, the algorithm will converge. Afterwards, we can retrieve this optimal set of weights to test if the weights learned help us to perform the task accurately. To do so, we can make a forward pass according to equation 5.5 with the optimal set of weights and test our loss function.

5.2.1 The simple pendulum classification problem

The simple pendulum is an idealization of a real pendulum. It consists of a point mass, m, attached to an infinitely light rigid rod of length l that is itself attached to a friction-less pivot point (Baker and Blackburn, 2005, p. 9). When the pendulum is displaced from its resting (equilibrium position), since no friction is assumed, this idealized pendulum will oscillate with a constant amplitude forever. Newton's second law, $F = mg$ provides the equation of motion for the pendulum:

$$ml\frac{d^2\theta}{dt^2} = -mg\sin\theta. \tag{5.6}$$

In equation 5.6, θ is the angular displacement of the pendulum from the vertical equilibrium and g the acceleration of gravity. The manipulation of this same equation, assuming both acceleration and length equal to 1 for simplicity, and $\ddot{\theta} \equiv \frac{d^2\theta}{dt^2}$, leads to:

$$\ddot{\theta} = -\sin\theta. \tag{5.7}$$

Equation 5.7 can be seen as a differential equation representing the motion of the pendulum since we would like to calculate how does the angle θ evolves across time, i.e. $\theta(t)$. By solving this second order differential equation, we can calculate such function given some initial conditions $(\theta_0, \dot{\theta}_0)$. $\dot{\theta}_0$ can also be viewed as the angular velocity at which we throw the pendulum. These inital conditions can be visualized in figure 5.5.

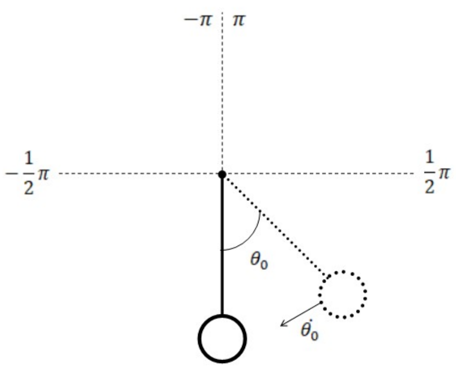

FIGURE 5.5: The simple pendulum graph.

If we allow our pendulum to be thrown from an angle greater than $\frac{1}{2}\pi$ in absolute value and we do not limit the initial angular velocity, two things can happen as time goes by:

- θ varies within a fixed interval without never being greater than π. In this case the pendulum would oscillate.
- θ eventually becomes greater than π at some point. In this case we will say that the pendulum "circulates".

These are the two classes we consider an initial position $(\theta_0, \dot{\theta}_0)$ may belong to. Figure 5.6 shows which initial pairs would lead to a pendulum oscillation and which ones to a pendulum circulation. The initial conditions $(\theta_0, \dot{\theta}_0) = (0, 0)$ lead to a third class, in which pendulum stays in equilibrium neither oscillating nor circulating. Practically speaking, the probability of having these as initial conditions is 0. That is the reason why our problem will not be a multi-class but a binary class classification problem. The same happens for those initial conditions in the separatrix, which at infinite time would lead to the unstable equilibrium where the pendulum stays at $-\pi$ or π without actually moving.

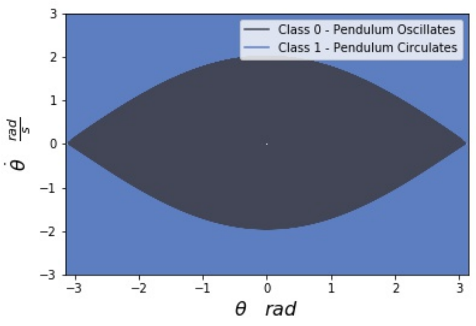

FIGURE 5.6: Initial conditions type as a function of θ and $\dot{\theta}$.

We will teach some neural networks to perform this task and compare its architectures. Apart from different neural network architectures, we will also use two two different types of training sets to train each network:

- θ_0 and $\dot{\theta}_0$ will be living in intervals $[-\pi, \pi]$ and $[-15, 15]$ respectively. We will retrieve n evenly spaced samples over each of the intervals and pair them together. This operation can be seen as a Cartesian product between both sets. Therefore, we will have n^2 samples in this approach. We will reference this training set as `GRID_nsquared`.

- In the second case, instead of making a grid of both parameters, we will retrieve n^2 values of θ from a uniform distribution in the interval $[-\pi, \pi]$ and another set of n^2 of $\dot{\theta}$ from the same distribution in the interval $[-15, 15]$. We will make pairs from this two sets to create our training set. We will reference this training set as `UNIF_nsquared`.

For both ways to create a training set, we will consider $n = 20$ and $n = 50$. The test set, though, will be fixed to 400 samples retrieved using the second approach. The aim of this design is to have a training set which actually knows the distribution of the data (`GRID`) and another training set which would be more realistic (`UNIF`, in which you don't have access to the whole distribution of the variable).

5.3 Adaptation of the algorithm

Once the problem has been exposed, we need to specify both the method to use and with which parameters. The decreasing *inertia weights* method with 20 particles seemed appealing to the problem, since, on average, it performed better than the others. We have to specify two additional elements, **the fit function** and the **number of parameters** and its ranges.

The fit or **objective function** will be a mean squared error loss (*mse*, equation 5.8). However, we may think of other fit functions that could perform similar jobs.

$$mse = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2. \tag{5.8}$$

To evaluate the performance of PSO for Neural Networks, we consider that studying how does the algorithm perform depending on the **number of parameters** is crucial because when facing a machine learning problem, one can think of designing several models that would fit their needs. Each of these models or architectures has a different number of parameters to optimize. Table 5.1 summarizes how is the nature of each of the architectures. For us, the third architecture would be big despite having only 18 parameters. We acknowledge that in the world of neural networks we can have architectures with millions of parameters to optimize. However, for the purpose of testing PSO, for us 18 parameters to optimize enough to be considered as "big".

Architecture	Parameters to opt.	Layers	Hidden layers	Hidden neurons
small	3	3	1	[1]
medium	9	3	1	[3]
big	18	4	2	[3,3]

TABLE 5.1: Nature of all architectures we considered.

In all cases, the input layer has two neurons (because we have two numerical inputs) and the output layer has one neuron (whether the sample belongs to class 1 or 0). For this reason, in all architectures, the activation function for the last layer will be a *sigmoid*, which outputs a number between 0 a 1. If the prediction for one sample is bigger or equal to 0.5, we will say the sample belongs to class 1. The vector of classes of all samples will be called $\hat{y}^{\text{bin}}$. On the other hand, due to common agreement, the activation function for all hidden layers will be set to *ReLu* activation. Finally, another common aspect between models is that the ranges of all weights will be $[-5, 5]$.

Provided these different models, we have to modify the specification of the **configuration text file**. To summarize, in table 5.2 we can see the specification of this file for each of the architectures. Notice we added several lines to be able to generalize for different structures. n stands for "none" r stands for ReLu and s stands for sigmoid:

Config. file lines	small	medium	big
Population size	20	20	20
Number of parameters	3	9	18
Weights range	$[-5, 5]$	$[-5, 5]$	$[-5, 5]$
Number of layers	3	3	4
Neurons layer 1 (input), f_{act}	2,n	2,n	2,n
Neurons layer 2 (hidden), f_{act}	1,r	3,r	3,r
Neurons layer 3 (hidden), f_{act}	-	-	3,r
Neurons layer 3 (output), f_{act}	1,s	1,s	-
Neurons layer 4 (output), f_{act}	-	-	1,s

TABLE 5.2: Configuration parameters for all architectures considered.

5.4 Tests an results

For each metric we consider we will present a separate table. In this case, we will recognize two different measures of performance:

- **The value of the loss function**, which can be calculated according to equation 5.8.
- **Accuracy measure**, that can be seen as the portion of a set of samples that have been classified correctly. It can be computed as:

$$acc = 1 - \frac{1}{N}\sum_{i=0}^{N} |\hat{y}_i^{\text{bin}} - y_i|. \tag{5.9}$$

In both cases, each number is the mean of 5 training process for a given training set and an architecture. Let's begin with the measure of the loss function:

Training data	small [2,1,1]		medium [2,3,1]		big [2,3,3,1]	
	train loss	test loss	train loss	test loss	train loss	test loss
GRID_2500	0.144	0.154	0.030	0.050	0.063	0.076
UNIF_2500	0.143	0.143	0.044	0.050	0.070	0.075
GRID_400	0.143	0.154	0.030	0.050	0.034	0.063
UNIF_400	0.137	0.143	0.040	0.050	0.039	0.050

TABLE 5.3: Results of different architectures (measured as the value of the fit function).

Several aspects of these results are worth mentioning. Recall that these results are just for the binary classification pendulum problem:

- The loss in the test set is higher than the training loss: this happens in all cases as expected. Weights are optimized exposing the algorithm to the training data, therefore, these weights are specific for this set of samples.
- 3 parameters are not enough to characterize our data: for the small architecture with just 3 parameters, we are not able to get a loss below 0.100 compared to the other two architectures. As a consequence, we may conclude that data is more complex than our model is.
- For medium and big architectures the value of the loss function is good enough: considering the minimum value one can get is 0, these values for the loss function seem not bad at all.

The other measure is the accuracy score:

Training data	small [2,1,1]		medium [2,3,1]		big [2,3,3,1]	
	train acc	test acc	train acc	test acc	train acc	test acc
GRID_2500	0.920	0.900	0.924	0.918	0.927	0.91
UNIF_2500	0.906	0.900	0.912	0.903	0.906	0.900
GRID_400	0.9300	0.900	0.930	0.900	0.930	0.925
UNIF_400	0.913	0.900	0.928	0.910	0.917	0.930

TABLE 5.4: Results of different architectures (measured as accuracy).

In this case, results are not good as expected because roughly 90% of the data belong to class 1. For this reason, these values contrast with the values of the fit function. In other words, we are minimizing the function correctly but we are not classifying the majority of samples whose initial conditions lead to an oscillation of the pendulum correctly.

A recent paper published, Nandi and Jana, 2019, explain that optimizing the weights of a neural network often leads to the problem of trapping in local minima with a very good convergence rate. This seems to fit our needs because, each time we train two equal neural networks, we have different optimal weights with very similar fit values. We can say, therefore, that we have multiple local minima.

In their paper, they propose another variant of PSO, PPSO. These variations would help to avoid the problem of being trapped in a local minimum. They claim that this method leads to a better accuracy and outperforms other variants of PSO. The modifications are very simple:

- Number of swarm particles must be set to 50.
- Maximum number of iterations is set to 500 instead of 100.
- c_1 and c_2 are changed from 2 to 1.6 and 1.7 respectively.
- Instead of decreasing inertia weights, we have a linear **increase** from 0.4 to 0.9 of such weights. In their paper, they state that $w(t)$ must be equal to:

$$w_{\text{min}} + \tanh(t \times \frac{w_{\text{max}} - w_{\text{min}}}{t_{\text{max}}}). \tag{5.10}$$

As you can see, every change is aimed to increase the balance between exploration and exploitation, as they claim in their paper. We will test this PPSO method with the medium architecture because, as you can see in table 5.4, the difference of results between medium and big architectures is meaningless and, on the other hand, with just 3 parameters we are not able to characterize our training set. Roughly speaking, an architecture with just three weights is not complex enough to characterize our data. On the other hand, if we increase a lot the number of parameters to be optimized, the fact that all of them are optimized simultaneously plays against it.

Training data	medium [2,3,1] LOSS		medium [2,3,1] ACC	
	train loss	test loss	train acc	test acc
`GRID_2500`	0.038	0.047	0.924	0.908
`UNIF_2500`	0.043	0.046	0.917	0.915
`GRID_400`	0.035	0.050	0.930	0.900
`UNIF_400`	0.039	0.046	0.925	0.915

TABLE 5.5: PPSO results for medium architecture for the binary classification problem of the pendulum.

As you can see, even by increasing the possibilities that each particle explores another region to escape from a local minimum, the training process does not perform noticeably different. It is true that accuracy have increased slightly, but not enough to talk about a major improvement. On the other hand, fit values are small enough to tell that PSO succeeds in minimizing the function, at least, locally. Further possible improvements to increase the performance of fitting our data with some weights are discussed in the conclusions.

Chapter 6

Conclusions

As suggested at the beginning, the Particle Swarm Optimization algorithm seemed promising but few applications have actually used it. For this reason, we started claiming that our thesis was aimed to exploit some of the potential of the method. To do so, the development of the library had to be undertaken and, once done, we could think of some real-world applications to test its performance.

Before actually developing both applications, studies regarding its performance in different functions have shown very interesting results. It has been noticed the excellent performance in functions such as the parabola or the multiple global minima function, but in more complex cases the algorithm converged to local minima in some executions. It is also important to mention that the computational time of an execution of the algorithm is fast enough so that a hundred of runs on a row doesn't imply more than a few seconds to finish. Further problems arose when we moved forward to real-world applications.

In both cases, further development had to be done because what actually worked for benchmark functions did not for these applications. For the former application, the Fourier analysis problem, parameters had to be refined one by one in order to make the algorithm optimize correctly the parameters because optimizing the set of parameters at the same time did not lead to pleasant results. With this methodology, the mean and deviation rates for the fit value decreased to a considerably good precision. Although the computational resources have augmented as the number of iterations performed by the algorithm have increased, the structure of the swarm and its particles have reduced complexity, leading to similar execution times when compared to the first approach results. On the other hand, results for the Neural Network weights optimization part were not as good as expected.

First of all, as already explained, the swarm get stuck in local minima although this did not happen for the benchmark functions. The main fact behind this behaviour is the assumption that at each iteration, we are approaching the global minimum with higher precision. As explained in the theoretical framework, the modification of the algorithm putting decreasing inertia weights was thought, precisely, to this aim. In contrast, since we have hundreds of data points whose label has to be predicted, optimizing for such number of points is challenging. In other words, if we change the whole set of optimal weights for another set very far away from the former, results could improve substantially. For this reason, the algorithm was changed to give more importance to exploration rather than exploitation. On top of that, the fact that all weights are optimized simultaneously does not allow the method to efficiently explore the space.

Putting this all together, it is clear that by making a more efficient exploration throughout the optimization process of the parameter space results could be improved. For this reason, deterministic optimization methods, which do not have these kind of problems, like gradient descent still are a better choice when optimizing neural network weights.

To summarize, one can conclude that Particle Swarm Optimization technique is a worthy candidate for function optimization procedures as long as the algorithm is adapted to the problem. In other words, for each optimization procedure in which we want to use PSO, the method has to be correctly adapted if we aim to have higher performances. Therefore, a lot of specific research should be done in the topic of interest in order to achieve better results, which makes the algorithm difficultly scalable.

Appendix A

Specific Contributions

As this thesis has been made by a group of two students, Albert Prat and Núria Valls, the contributions of each of the members will be specified below.

From the beginning of the thesis, both of us started a research about the Particle Swarm Optimization technique. Once obtained a little bit of context, we splitted our work into two tasks. On the one hand, Albert started to learn more about the state of the art of PSO implementations and improvements, which has ended up being Chapter 2. On the other hand, Núria started to implement a basic PSO algorithm from scratch supported by the notes from Albert on the theoretical part. This implementation was explained by Núria in the first two sections of Chapter 3. Also in the same chapter, Albert was in charge of doing the parametric study of the algorithm implemented, whereas Núria was in charge of the tests and results section.

Chapter 4 has been entirely made by Núria, starting from the comprehension of the problem, implementation of the adaptation of the algorithm and evaluation of the results, up to the implementation of the improved approach and its corresponding performance evaluation.

Finally, Chapter 5 has been entirely made by Albert following a similar structure than the previous chapter. First of all with the definition of the problem, followed by the implementation of the algorithm adaptation and the evaluation of the results obtained.

Since we consider Chapters 1, 2 and 3 to be necessary in order to understand further sections of the thesis, the whole original document without cuts have been included despite having it split as exposed above. On the other hand, only one of the following chapters has been summarized: Chapter 4, 5. In this thesis, the latter Chapter has been included in its whole.

Appendix B

GitHub Repository

All the code regarding the implementation of this thesis is available in a public GitHub repository which can be accessed through following link to PSO and two real world applications.

Bibliography

Ackley, D.H. (1987). “A connectionist machine for genetic hillclimbing”. In:

Adewumi, A.O. and A.M. Arasomwan (2015). “Improved Particle Swarm Optimizer with Dynamically Adjusted Search Space and Velocity Limits for Global Optimization”. In: *International Journal on Artificial Intelligence Tools* 24.5. DOI: 10.1142/S0218213015500177.

Baker, Gregory L. and James A. Blackburn (2005). *The pendulum: a case study in physics*. Oxford University Press.

Bratton, Daniel and James Kennedy (2007). “Defining a Standard for Particle Swarm Optimization.” In: *2007 IEEE Swarm Intelligence Symposium, Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, p. 120. ISSN: 1-4244-0708-7.

Clerc, Maurice (1999). “The swarm and the queen: towards a deterministic and adaptive particle swarm optimization.” In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, p. 1951. ISSN: 0-7803-5536-9.

Domingos, Pedro (2012). “A Few Useful Things to Know About Machine Learning”. In: *Commun. ACM* 55.10, pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/2347736.2347755. URL: http://doi.acm.org.sire.ub.edu/10.1145/2347736.2347755.

Eberhart, Russell C. and James Kennedy (1995). “A new optimizer using particle swarm theory.” In: *MHS’95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Micro Machine and Human Science, 1995. MHS ’95., Proceedings of the Sixth International Symposium on*, p. 39. ISSN: 0-7803-2676-8.

— (1997). “A discrete binary version of the particle swarm algorithm.” In: *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, p. 4104. ISSN: 0-7803-4053-1.

Eberhart, Russell C. and Yuhui Shi (2001a). “Particle swarm optimization: developments, applications and resources.” In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), Evolutionary Computation, 2001. Proceedings of the 2001 Congress on, Evolutionary computation*, p. 81. ISSN: 0-7803-6657-3.

— (2001b). “Tracking and Optimizing Dynamic Systems with Particle Swarms”. In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546*.

Engelbrecht, Andries (2012). “Particle Swarm Optimization: Velocity Initialization”. In: *2012 IEEE Congress on Evolutionary Computation*, pp. 1–8. DOI: 10.1109/CEC.2012.6256112.

Feng, Yong et al. (2007). “Chaotic Inertia Weight in Particle Swarm Optimization”. In:

Frigo, Matteo and Steven G. Johnson (2005). “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2. Special issue on “Program Generation, Optimization, and Platform Adaptation”, pp. 216–231.

Gomez, Gerard, Josep-Maria Mondelo, and Carles Simó (2010). “A collocation method for the numerical Fourier analysis of quasi-periodic functions. II: Analytical error estimates”. In: *Discrete and Continuous Dynamical Systems. Series B* 1. DOI: 10.3934/dcdsb.2010.14.75.

Jacob, Christian J. et al. (2007). “"SwarmArt": Interative Art from Swarm Intelligence”. In: *Leonardo* 40.3, pp. 248–255.

JetBrains (2019). *CLion*. URL: https://www.jetbrains.com/clion/.

Kennedy, James and Russel Eberhart (1995). “Particle swarm optimization”. In: *IEEE International Conference on Neural Networks*. Vol. 4, pp. 1942–1948.

Kriesel, David (2007). *A Brief Introduction to Neural Networks*. URL: availableathttp://www.dkriesel.com.

Laskar, Jacques (1999). “Introduction to Frequency Map Analysis”. In: *Hamiltonian Systems with Three or More Degrees of Freedom*. DOI: 10.1007/978-94-011-4673-9_13.

Marini, Federico and Beata Walczak (2015). “Particle swarm optimization (PSO). A tutorial”. In: *Chemometrics and Intelligent Laboratory Systems* 149, pp. 153–165.

Millonas, Mark M. (1994). “Swarms, Phase Transitions, and Collective Intelligence”. In: pp. 137–151.

Montalvo, Idel et al. (2008). “Particle Swarm Optimization applied to the design of water supply systems”. In: *Computers and Mathematics with Applications* 56, pp. 769–776.

Nandi, Arijit and Nanda Dulal Jana (2019). “Accuracy Improvement of Neural Network Training using Particle Swarm Optimization and its Stability Analysis for Classification”. In: *CoRR* abs/1905.04522. arXiv: 1905.04522. URL: http://arxiv.org/abs/1905.04522.

Ritchie, Dennis M. (1993). “The Development of the C Language”. In: *SIGPLAN Not.* 28.3, pp. 201–208. ISSN: 0362-1340. DOI: 10.1145/155360.155580. URL: http://doi.acm.org/10.1145/155360.155580.

Rosenbrock, H. H. (1960). “An Automatic Method for Finding the Greatest or Least Value of a Function”. In: *The Computer Journal* 3.3, pp. 175–184. DOI: 10.1093/comjnl/3.3.175. URL: https://doi.org/10.1093/comjnl/3.3.175.

Shi, Yuhui and Russell C. Eberhart (1998a). “A modified particle swarm optimizer.” In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360), Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, p. 69. ISSN: 0-7803-4869-9.

— (1998b). “Parameter selection in particle swarm optimization”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1447, pp. 591–600.

Talukder, Satyobroto (2011). “Mathematical Modelling and Applications of Particle Swarm Optimization”. MA thesis. Blekinge Institute of Technology.

Wilson, Edward O. (1975). *Sociobiology: The Abridged edition*. The Belknap Press.

Xiaohui, Hu, Yuhui Shi, and Russel C. Eberhart (2004). “Recent advances in particle swarm.” In: *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753), Evolutionary Computation, 2004. CEC2004. Congress on, Evolutionary computation*, p. 90. ISSN: 0-7803-8515-2.

Xin, Jianbin, Guimin Chen, and Yubao Hai (2009). “A particle swarm optimizer with multi-stage linearly-decreasing inertia weight”. In: *International Joint Conference on Computational Science and Optimization* 1, pp. 505–508.

Zhang, Fengge et al. (2018). “Rotor Optimization Design of Brushless Doubly Fed Machine Based on Improved Particle Swarm Optimization.” In: *2018 21st International Conference on Electrical Machines and Systems (ICEMS), Electrical Machines and Systems (ICEMS), 2018 21st International Conference on*, p. 564. ISSN: 978-89-86510-20-1.