Machine Learning

Alexandre Castelnau

Fall 2021

Georgia Institute of Technology

Table des matières

1 Introduction

As Yurii Nesterov said *"Whatever people do, at some point they get a craving to organize things in a best possible way. This intention, converted in a mathematical form, turns out to be an optimization problem of certain type"*. This quote emphasizes the importance and omnipresence of the notion of optimization. Also, we know there is many important theorems giving clues about the existence (or even sometimes the uniqueness) of an optimum under some hypotheses (twice Fréchet differentiable function on an open neighborhood with a null gradient...) but often computing or finding these optimum is still an hard problem. That is why, we developped algorithms that perform an exploration of the space we consider in order to find some optimum.

Through this assignment, we will explore how we perform random search. That is why, we will apply the most common randomized optimization algorithms (*Random Hill Climbing*, *Simulated Annealing*, *Genetic Algorithm* and *MIMIC*) to three classical optimization problems (*OneMax*, *Knapsack* and *Four Peaks*). These applications of optimization algorithms will allow us to highlight their characteristics (advantages, defects, problems where their application is preferable...).

Later, we will focus on the application of these methods in a framework that is much more interesting for us given the course we are taking : the optimization of the weights of a neural network.

2 Discovery of different optimization algorithms

As announced above, we will deal with the following problems : *OneMax, KnapSack* and *Four Peaks*. For each of them, we will discuss the interest that the problem may have in our quest for characterizing algorithms before discussing and analyzing our results.

2.1 OneMax

2.1.1 Description of the problem

The first problem we consider is the very simple *One Max optimization problem.*

We consider a n-dimensional binary vector x and the fitness function is defined as :

$$F(x) = \sum_{i=0}^{n-1} x_i$$

I have chosen this first problem because it presents several interesting characteristics that can be perceived immediately. First of all, by observing the fitness function, one can quickly see that there is a unique global optimizer (the vector $\mathbf{1} = (1)_{0\leq i<n}$). To express another characteristic of the problem, we define the neighborhood of a vector x this way :

$$N(x) = \left\{ y = (y_i)_{0\leq i<n} | \sum_{i=0}^{n-1}(x_i \oplus y_i) = 1 \right\}$$

It is the set of all n-dimensional vector that differ only by one bit (XOR operation) of the vector x. With this definition, we quickly understand that for any vector, different than the global optimizer $\mathbf{1}$, it exists a vector y such that $F(y) > F(x)$. Hence, the space we considering, with this fitness function, only contains one global optimizer and no other local maximum. Indeed, if x is not $\mathbf{1}$, it exists i such as $x_i = 0$ and vector y, such as all its bits except the y_i are equal to those of x, is in $N(x)$ and satisfy the relation $F(y) > F(x)$.

With this knowledge, we can be sure that an algorithm such as *Random Hill Climb (RHC)* will perform well. Indeed, during a iteration of *RHC*, this algorithm choose a point in the neighborhood of the current state and update it if this new state is better than the current one. The current fitness value can only go up and get stuck in a local maximum. Here, there is no trap for *RHC* (non-global maximum) and it will exploit its full potential.

2.1.2 Implementation and analyse

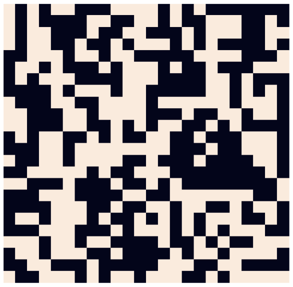


Figure 1 – Initial vector (24 × 24 format)

First, because it seems to me that it was a very simple problem, I choose to consider 576-bits long vector. When it was needed (for *RHC* and *Simulate Annealing (SA)*), I provided an initial vector (Figure 1) randomly chosen such as 1 and 0 as the same odds to appears (uniform distribution).

Then, with the use of *MLRose Hiive Python package*, I was able to implement the different optimization algorithms and compare them. Figure 2 is what I was able to plot. We can observe only 3 of the 4 algorithms we are considering.

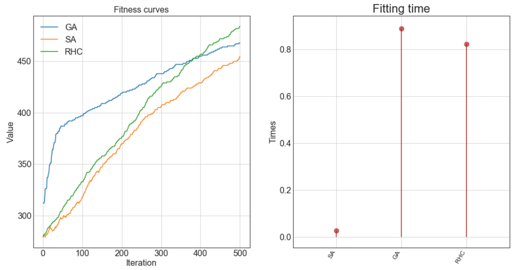


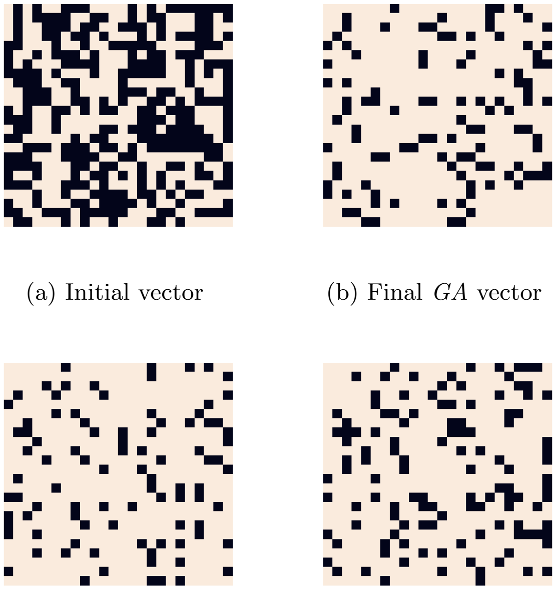
Figure 2 – Performances on the 576-bits long OneMax problem

This absence is because *Mimic (MM)* was not able to converge in a reasonable time (running for hours without answer). But we can already get some interesting insights. As we said previously, this problem have a particular shape that allows *RHC* to outperform the other algorithm (only one local maximum, the global one). Meanwhile, it is notable that *Genectic Algorithm (GA)* have an outstanding behaviour on this problem too. For fewer iterations of the algorithm, *GA* was way above the two other and at the end have similar result than the *RHC*. Indeed, I run this same problem with several seed, results from *RHC* and *GA* were always close and sometimes *GA* was even way above *RHC* with the use of a "lucky start" (when the starting population of the *GA* contains good candidate states - high fitness value). By the way, the time used by both algorithms is quite close.

In Figure 3, we represent the final state of each algorithm to see how all algorithms were able to perform optimisation.

Also, to complete the exploration of this kind of problem, I tried to do the same problem but with a vector having a smaller dimension in order to have the performance of the *MM*.

We can observe once again in Figure 4 how RHC performs well on the OneMax optimization problem but also that *MM* also returns the best state possible and seems to return it a fewer itereation.



(a) Initial vector

(b) Final *GA* vector

(c) Final *RHC* vector

(d) Final *SA* vector

Figure 3 – Important vector states

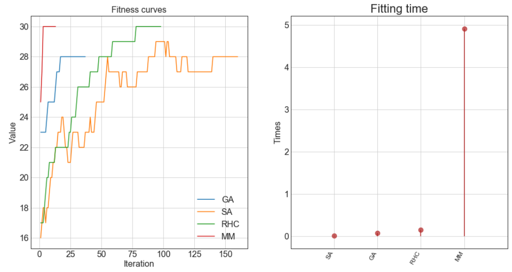


Figure 4 – Performance on 30-bits vector

Meanwhile, we must be careful when we analyze the first chart, the *MM* seems to converge with less iteration but it is not necessarily the case. Indeed, the attribute fitness curve that we use to draw it returns an array of couple of values. If the first value is the fitness evaluation of the state during the iteration, I was not able to determine the meaning of the second value by reading the code of the *MLRose Hiive* modules, so I plotted the fitness curves according the number of pairs of values that the attribute returned (this is true for all curve I plotted during the assignment). That is why each time, the time needed to perform the optimization is present because we don't really care about the number of iteration but knowing the scalability of each algorithms is important. Here (and it will also be the case for the other problems) *MM* is way slower than the other especially RHC who gives the best state too.

2.2 Four Peaks

2.2.1 Description of the problem

The *Four Peaks* optimization problem considers n-dimensional state vector and has the following fitness function :

$$F(x,T) = \max(tail(0,x),\ head(1,x)) + R(x,T)$$

where $tail(0,x)$ is the number of successive 0 at the end of x, $head(1,x)$ the number of successive 1 at the beginning of x and $R(x,T)$ is equal to n if tail and head are over the threshold value T or else is equal to 0.

In this discrete optimization problem, we are still considering the same definition of the neighborhood for a vector x :

$$N(x) = \left\{ y = (y_i)_{0 \leq i < n} | \sum_{i=0}^{n-1} (x_i \oplus y_i) = 1 \right\}$$

With this information and a careful reading of the fitness function, we can observe several things. There are four different local maximum (I assume, without a doubt, that this is the reason this problem has a such name) : vectors $\mathbf{1}$, $\mathbf{0}, X_1$ and X_0. Indeed, if we take $\mathbf{1}$, we know that $F(\mathbf{1}) = n$ and any $y \in N(x)$, it exists i such as $y_i = 0$ and due to that $F(y) \in \{i-1, n-(i-1)\} < F(\mathbf{1})$. For $\mathbf{0}$, we can think in a similar way.

Also, there are only two of them that are global optimizers : X_0 and X_1 where X_1 is the vector with almost full of 1 (except the $T+1$ last bits) and X_0 built in a similar way (X_1 and X_2 are optimizers because : for all $y \in N(X_1)$, $F(y) < n-T < F(\mathbf{1}) = n < 2n-T-1 = F(X_1)$ and we have same inequalities if X_2 takes the place of X_1 and $\mathbf{0}$ replace $\mathbf{1}$).

Hence, the existence of these local and global optimizers will enable us to test the capacity of each algorithm to not be stuck in the influence sphere of a local maximizer.

2.2.2 Implementation and analyse

With the knowledge that *MM* can be time expensive with big state vector, I decide to test the algorithms on a 30-bits long vector and I set the threshold at 3. Hence :

$$X_0 = 111100\ldots00 \text{ and } F(X_0) = 56$$
$$X_1 = 11\ldots110000 \text{ and } F(X_1) = 56$$

Once again, for *RHC* and *SA*, I provided an random initial vector (chosen with uniform probability).

On the first chart of Figure 5, we can clearly see that *RHC* performs poorly. Indeed, *RHC* can get stuck in the neighborhood of a local optimizer and Four Peaks highlights this fact with the presence of **1** and **0**. Also, we observe that *GA* and *MM* performs in a similar way in this problem. It is quite normal because those two algorithms gather some similar characteristics on how they work (use of the best element of a population...). They both have a good approximation of the optimizer but were not able to reach it. It can be due to the way they cross the population *GA*, the modification between each iteration may be too important to detect improvement during the last step of the "convergence".

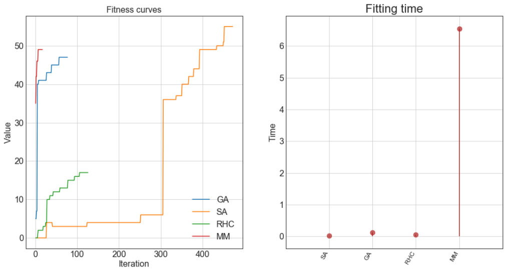


FIGURE 5 – Performances on 4 Peaks

But, we can see that *SA* is the best kind of algorithm for this problem (it returns one of the two best state). This result is not surprising. We start with a randomly chosen vector (which has no reason to be particularly close to a local maximum) and we still evolve in a space with few local maximums. Moreover, SA allows "going back" in case it would explore a neighborhood of a local maximum. All these elements (few trapping maximums coupled with the possible backtracking) explain why *SA* has a notable performance on this problem, with a correct running time.

With this problem, we can try to explore some others characteristics of *SA* as the influence of the decay schedule, its robustness to numerical errors... Figure 6 highlights differences of evolution between each schedule possible, but we can observe that all seem to converge to a state close to an optimal one (we will see later that it is not the parameter in the schedule having the best influence on the convergence of the algorithm).

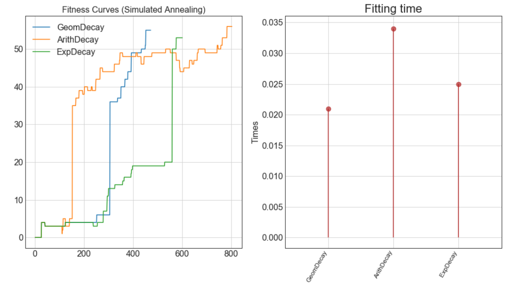


FIGURE 6 – Influence of Decay Scheduling

Then, with the presence of the four peaks, testing the ability of SA to go out of the neighborhood of a local optimizer ($\mathbf{1}$ or $\mathbf{0}$) to reack the neighborhood of a global one (X_1 or X_2) seems to be a good ideas. For that, I tried to observe the evolution of the fitness curve when the initial state was "bad". I produced this situation with the following initial vectors :

$$x_0 = (100\dots00) \ \in N(\mathbf{0})$$
$$x_1 = (11\dots110) \ \in N(\mathbf{1})$$

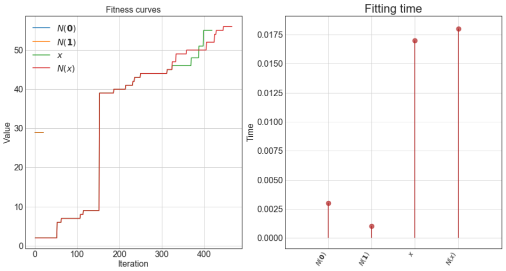


FIGURE 7 – Robustness of SA and worst case scenario

In Figure 7, we can observe the result of this little experimentation. It appears that with default scheduling parameters, $\bullet$ cannot escape the nearest neighborhood of a local optimum. Also, this figure shows the robustness of SA to numerical errors. What the meaning of this kind of robustness ? It is when we try to show that :

$$y \in N(x) \text{ and } SA(x) \underset{\infty}{\to} b \Rightarrow SA(y) \underset{\infty}{\to} b$$

To test it, I took two different vectors, one totally randomly chosen and the other one with one bit of the first flipped. With the chart plotted, we can observe that SA is quite robust to numerical errors.

Let's go back to the fact that SA is unable to exit a local maximum with the default settings. We must remember that SA has the characteristic of being able to "going back". This characteristic is due to the fact that we define a probability of "going back" according to the value of the neighbor. This probability is parameterized (in other) by the initial value of the temperature. For instance, for an geometrical decay, this probability is defined at the time t as :

$$T(t) = \max(T_0 \times r^t, \ T_{min})$$

The probability of considering a neighbor with a lower value is all the more important as the initial temperature is large. In our worst case scenario, the possibility of "going back" will be useful several times, so large value for T_0 can be useful.

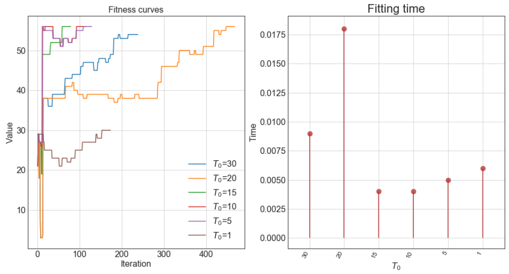


FIGURE 8 – Influence of initial temperature

In Figure 8, we see clearly the influence of this parameter and it can provides us the ability to escape of "traps" and reach the neighborhood of a global optimizer. Meanwhile, even this experimentation don't show this, we must understand that there is a tradeoff in the choice of T_0. Indeed, high values provides the ability to not be stuck in a local optimum but too important values will not be useful for SA because each iteration will appear as random one.

2.3 KnapSack

2.3.1 Description of the problem

The *KnapSack* is a well-known optimization problem (one of the *Karp's 21 NP-complete problems*). It considers a set of n items where $item_i$ has a weight w_i and a value v_i and we also have a maximum knapsack capacity W. This problem is the mathematical form of the problem of space allocation when we prepare our bagback or a problem of portfolio optimization... The fitness function defined by the problem is :

$$F(x) = \sum_{i=0}^{n-1} v_i x_i \text{ if } \sum_{i=0}^{n-1} w_i x_i \leq W \text{ and } 0 \text{ otherwise}$$

This problem is much more complicated than the previous ones. First of all, for a fixed dimension, the problem also depends on the weights and values that we determine. Thus the space of values that the fitness function takes may be much more complex (it is most of the time) than for the previous problems. Indeed, an exhaustive analysis of the space is much more complicated but we can say without any doubt that there is a multitude of local optima. Thus, algorithms such as SA and RHC are likely to encounter more difficulties than in the other problems and algorithms such as *MM* and *GA* will be more likely to outperform the two others.

2.3.2 Implementation and analyse

For the implementation of this problem, I considered a situation where we have the choice of 25 different items with a weight between 5 and 25 and an importance value between 1 and 5. I made a random assignment of each of these values and I put that we could keep only 60% of the total weight of all the items. For *RHC* and *SA*, I provided an initial null vector.

Here (Figure 9) are the optimization performances I obtained with the four algorithms. We clearly find that *MM* and *GA* are the best kind of algorithms for this problem. Let's try to understand why they outperform *SA* and *RHC*.

We have already discussed how *RHC* and *SA* explore space but let's make a quick analogy of the methodology for this problem. *RHC* starts with an empty suitcase and at each step it adds an item to the suitcase until it is blocked by the weight limit.

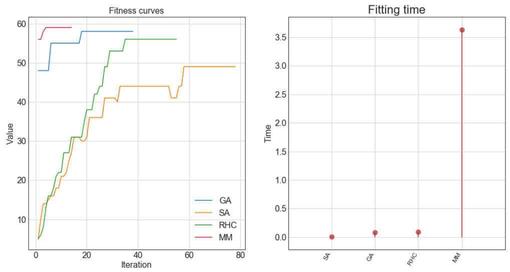
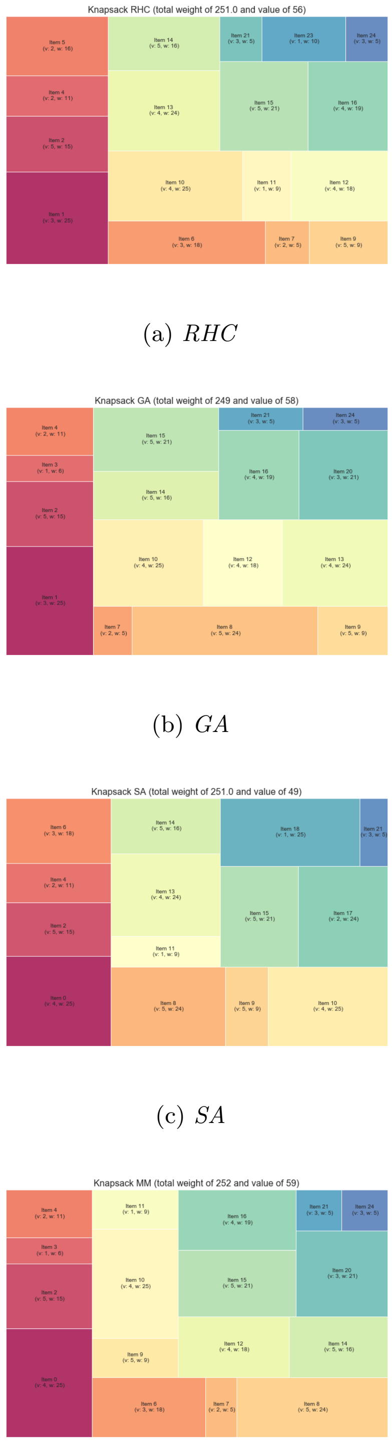


FIGURE 9 – Performances on *Knapsack*



(a) *RHC*

(b) *GA*

(c) *SA*

(d) *MM*

FIGURE 10 – Final States on the Knapsack problem

SA will also follow a similar process but will also occasionally remove an item. As for *GA* and *MM*, they use "populations". For example, for *GA*, it is like doing a set of trials to fill the bag, looking at whether the result is satisfactory and only using the best trials (by combination) to make the new set of trials. Whereas *MM* follows a similar methodology, with selection occurring as a result of density estimation. Due to the existence of multiple local optimum, *RHC* and *SA* are outperformed by *GA* and *MM* as we can see. Also, the two last algorithms needs less iterations to find a solution near the best (even it is more time consuming).

In figure 10, we can find all the combinations of items proposed at the end of the different algorithms (with their total weight and the cumulative value of the items).

3 Application of optimization methods in the context of Machine Learning

Now that we have done a first analysis of the four algorithms on classical optimization problems in order to detect their characteristics, we are going to look at their application to a domain that concerns us more directly : Machine Learning and the optimization of weight coefficients in a neural network. Often when we look for the optimal weights of the network, we use the backpropagation (and thus the gradient descent method applied on the error or the perceptron rule).But we will try to see the performance of the algorithms we have tested and try to compare the results with the results we get with the usual method.

3.1 Quick reminder of our problem

I took the dataset used during the first assignment on hospital mortality. This dataset includes 1177 tuples. Each of these tuples compiles dozens of attributes on each patient. Among them, the binary attribute "outcome" representes the data that we will have to predict (0 : patient lives, 1 : patient dies). It is a pure binary classification problem.

During the last assignment, we plotted the learning tied with the Neural Network (*MLP Classifier* with *SKLearn*). We have seen that adding new data point in the training set was not sufficient to have an impact on the accuracy score (see Figure 11, by the way here cross-validation and training score don't converge, epitome of a kind of overfitting). Since adding new data don't change the accuracy score, let's try to change the way we update the weights of our network to see if the model can be better.

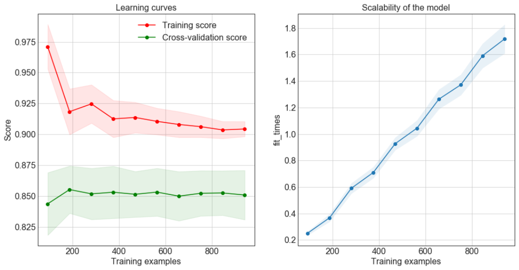


FIGURE 11 – Learning Curve of the *MLP Classifier*

3.2 Optimization methods applied to Neural Networks

Using the *Neural Network* object of the *MLRose Hiive* package, it is possible to implement a neural network and to set parameters to optimize it (maximum number of iterations, activation function of the elementary neuron, weight optimization algorithm...).

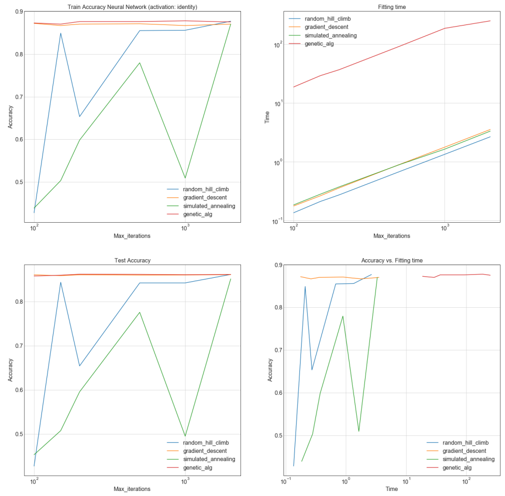


FIGURE 12 – Identity activation function

It is noteworthy that it is possible to perform an optimization other than with the "perceptron rule" and "gradient descent rule" with backpropagation and that in the following the "Gradient descent rule" curve will serve as a reference curve for the future comparison of the previously used and analyzed algorithms. Moreover, we can note that it is not possible to use a weight optimization with *MM*. Indeed, the optimization of the weights of a neural network is an optimization problem using continuous values. While it is relatively easy to determine the best elements of a population to combine them (for *GA*), it is much more complicated to determine a density estimate. Indeed, when evolving with continuous values, the probability of an event is often described as an integral of a probability density, but defining such a probability density from a finite set of points (the population) makes no sense.

Also in the context of a continuous value optimization problem we can define quite easily, a definition of the neighborhood that we can implement numerically. For example, let consider $x = (x_i)_{0 \leq i < n}$ such as all x_i are elements of $\mathbb{R}$, we can define the neighborhood of this vector this way :

$$N_\epsilon(x) = \{y = (y_i)_{0 \leq i < n} \ : \ ||x - y|| \leq \epsilon\}$$

where $||x - y||$ can be any function satisfying the definition of a distance and ϵ is a small real strictly positive value. The most common distance are the Euclidean distance ($\sqrt{\sum_{i=0}^{n-1}(x_i - y_i)^2}$) or the Manhattan distance ($\sum_{i=0}^{n-1} |x_i - y_i|$). This definition is the definition of the closed ball having x as center.

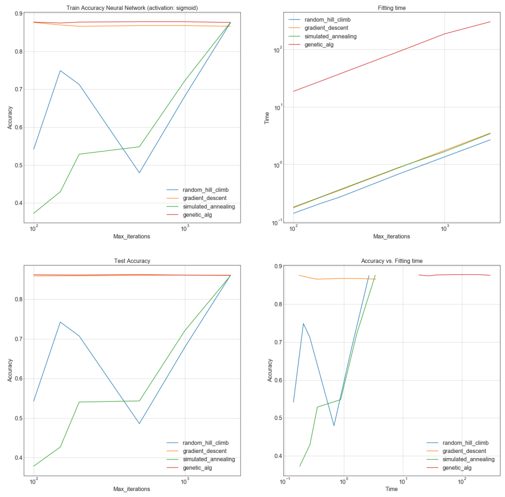


FIGURE 13 – Sigmoid activation function

Hence, during the exploration, we can easily compute neighbors of x. For instance, the set $V_\epsilon(x)$ such as :

$$V_\epsilon(x) = \{y = (x_0, \ldots, x_i + \eta, \ldots, x_{n-1}) \mid i \in [0, n-1], |\eta| \leq \epsilon\}$$

is a subset of $N_\epsilon(x)$ with an infinite cardinal and and we can easily get elements of this subset and so element of the neighborhood of x.

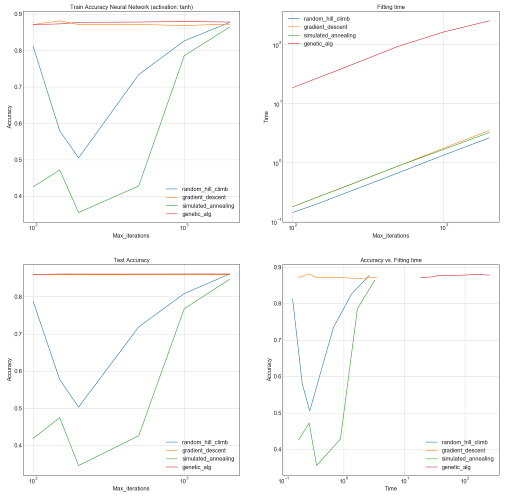


FIGURE 14 – Tanh activation function

Through figures 12 to 15, I have plotted for each possible activation function in the neural network (*relu*, *tanh*, *identity* and *logistic sigmoid*) four differents charts : train accuracy score, test accuracy, fitting time (all according the maximum allowed iteration) and the test accuracy (according the fitting time). All values of all chart are the mean of several optimizations of the weights having the same set of parameters (but not the same random state).

I will not go through each of these figures to describe them exhaustively, as they all have similar looks, but make a summary of what can be perceived.

Regardless of the activation function considered, *GA* and *Gradient Descent (GD)* have similar accuracy scores on training and testing sets which, moreover, are much higher than those obtained for the *RHC* or *SA* algorithms. Also, the latter require a much larger number of iterations to achieve a suitable accuracy on the training and testing sets. This confirms the analysis made in the first part. *GA* is much better adapted to provide a good approximation of a global optimum than the simple *RHC* and *SA* algorithms when the complexity of the optimization problem increases. However, the optimization of the continuous weights of a neural network falls within the scope of complicated problems, so it was expected that the accuracy of *GA* would be superior to the others.

However, by observing the tables on the top right and noting the logarithmic scale, we can also note that the fitting time of *GA* is much higher than the other algorithms (especially the fitting time of *GD*).

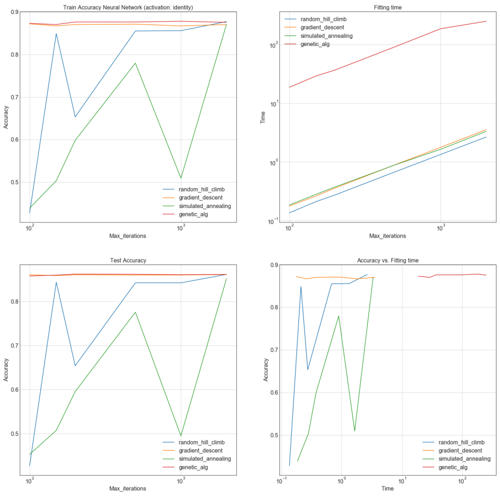


Figure 15 – RELU activation function

Therefore, for a similar accuracy, we will choose the least time consuming optimization algorithm. Here, it is the *GA*. The fourth graph of each figure could be read as a comparison of the algorithms. Indeed, the closer the curve of the algorithm was to the upper left corner of the graph, the more interesting the algorithm was. All these curves support the use of the *GD* (which in the context of backpropagation is the most widely used and documented algorithm) in the context of neural network optimization.

Moreover, by comparing the values obtained in the two graphs on the right, we can always see that a gap between the accuracies on the training and the testing set persists. This is the epitome of the presence of an overfitting phenomenon in each model.

4 Conclusion

Let's try to summarize the information we have gathered during these different analyses.

The evaluation of an optimization algorithm can be done on different criteria :

- convergence speed, time efficiency
- robustness to numerical errors
- and of course, the ability to approach a global optimizer

As we have observed in the context of optimization problems, depending on the formulation of the latter, the space that the algorithm will explore can be more or less complicated (for example : that of OneMax is much simpler than that of KnapSack). We can note that in the problems of the first part, we have mainly evoked the presence (or absence) of multiple local optima to define the complexity of the space, but other criteria can be taken into account (for instance what values we consider : integer, binary, continuous values...).

We could note that on the simplest problems (*One Max* or *Four Peaks*), algorithms such as *RHC* or *SA* can be efficient in terms of evaluation of the fitness function but also they are able to return an answer in a short time. On these problem, *GA* and *MM* also performed well but are way slower. By the way, for simple problem, we don't have to really car about the robustness to errors because there are few local optimizers, algorithms will mostly be able to converge to the same point if we let them enough iteration. For more complex optimization problem (such as KnapSack), algorithms as *MM* or *GA*, are a better choice. Indeed, their slower speed is largely compensated by their ability to reach the neighborhood of a global optimizer. Also, by the way these algorithms work, they are less likely to be stuck in one of the many local optimum and are robust to numerical error (because each iteration is defined by a population and not a single vector).

Now, when we think about the application of optimization algorithm to the Machine Learning problem "Finding the best set of weights for a neural network", we must directly think : "this is a hard problem, *RHC* and *SA* might have serious difficulties".

The Table 1 summarizes all the thing we could think about during this paper.

Algorithm	Advantages	Drawbacks
Random Hill Climb	Conceptually simple algorithm, easy exploration of the neighborhood, evaluation of the fitness function always in progress, works very well for simple problems with few local optimum, low fitting time, can be used on discrete or continuous optimization problem	Unsuitable for complex problems (can get stuck in the neighborhood of a local optimum), high number of iterations
Simulated Annealing	Conceptually simple algorithm, easy exploration of the neighborhood, works very well for simple problems with few local optimum, low fitting time, more flexible than RHC by allowing backtracking, can be used on continuous or discrete optimization problem	Result more variable with complex problems, not really useful for optimization of a neural network, high number of iterations
Genetic Algorithm	Good results in all types of problems, can be used on continuous or discrete optimization problem, having a new population at each iteration is relatively simple, robustness to numerical errors	Higher fitting time than RHC and SA, low number of iterations
Gradient Descent	Performs well on optimization of a neural network, is great for continuous optimization problem, good scalability	Can be stuck in local optimum
Mimic	Good results on complex problem, robustness to numerical error	Worst scalability (highest fitting time), can not be used on continuous optimization problem, understanding of the algorithm more complex, low number of iterations

Table 1 – Summary of all algorithms