TP5: NN and PSO

by Martino Ferrari

1 Artificial Neural Network

In this TP we will train an *Artificial Neural Network* (**NN**) to recognise some handwritten digits.

NN.s were introduced in the 50's to model some of the most elementary brain behaviours. A **NN** is a set of *Artificial Neurons*, organised in layers, interconnected together.

An *Artificial Neuron* is a very simple unit that applies a non-linear transformation to its input. As transformation, the *sigmoid* function, $g(x) = \frac{1}{1+e^{-x}}$ shown in figure 1, is often used.

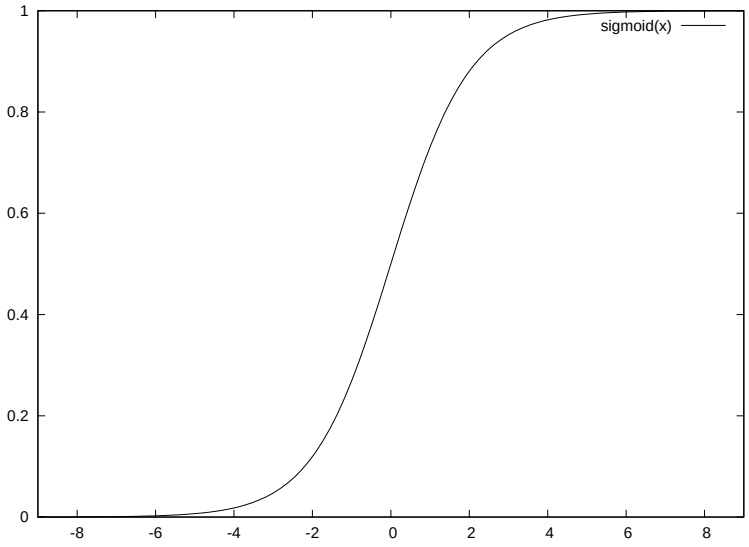

Figure 1. *sigmoid* function

The organisation in multiple layers is very important, each *neuron* of a layer is connected to all the *neurons* of the following layer with arcs of different weights, as shown in figure 2. It's important to notice note that the first layer is the **input** layer, the last is the **output** layer, and the middle ones (1 or more) are the **hidden** layers.

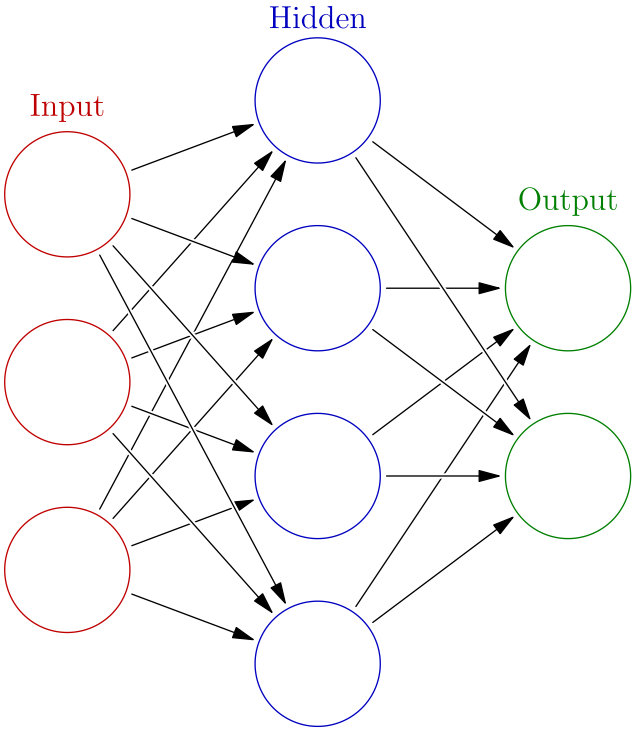

Figure 2. Simple **NN** (source: wikipedia)

The weighted arcs between two layers i, j can be represented as a matrix $\theta_{i,j}$ of size $n_i \times n_j$, where n_i and n_j are the number of neurons of the layer i and j, respectively.

In this way it's possible to represent a **NN** as a series of matrix multiplications and non-linear transformations, $x \to h_\theta(x) \to y$, with x the input and y the output.

The learning process of a **NN** is nothing else than a fine tuning of the different $\theta_{i,j}$ matricies to minimise the error $e(x_t, y_t)$ between the transformation of the training set x_t and the expected output y_t, $e(x_t, y_t) = \|h_\theta(x_t) - y_t\|$.

This error function will be our fitness function and the solution space $S \subset \mathbb{R}^n$, with n the total number of parameters to tune (this means that we will vectorize the $\theta_{i,j}$ matricies in a single vector $\vec{v} \in \mathbb{R}^n$). As it's clear from the definition, S is infinite and continue.

2 Particle Swarm Optimisation

The *Particle Swarm Optimisation* (**PSO**) is a metaheurstic introduced in 1995 by Kennedy, Eberhart and Shi and inspired to the **AS** (previously introduced). As the **AS**, the **PSO** is as well a swarm intelligence method but is minded to explore a solution space $S \subset \mathbb{R}^n$, as the case of the **NN**.

The **PSO** explores S using a *population* of m particles that uses both personal knowledge (*individuality)* and group knowledge (*sociality*) to find an optimal solution.

The particles will be guided at the same time by the group leader, their intuition and from their current movement.

Each particle i, at a time t, is represented by its his position $x_i(t) \in S$ and its velocity $v_i(t) \in R^n$. Both position and initial speed are randomly initialised.

At every iteration a new speed $v_i(t+1)$ and position $x_i(t+1)$ are computed as following

$$v_i(t+1) = \omega \cdot v_i(t) + c_1 \cdot r_1 \cdot (b_t^i - x_i(t)) + c_2 \cdot r_2 \cdot (b_t^G - x_i(t))$$

$$x_i(t+1) \xrightarrow{U} x_i(t) + v_i(t+1)$$

where:

- b_t^i is the current best solution found by the i-th particle
- b_t^G is the current best solution of the group
- ω is the *inertia* constant
- c_1 and c_2 are the *cognitive* and *social* parameters, controlling how much the particles will follow their intuition or the group leader
- r_1 and r_2 are random numbers between 0 and 1.

In figure 3 is possible to see a simple representation of the concept expressed before.

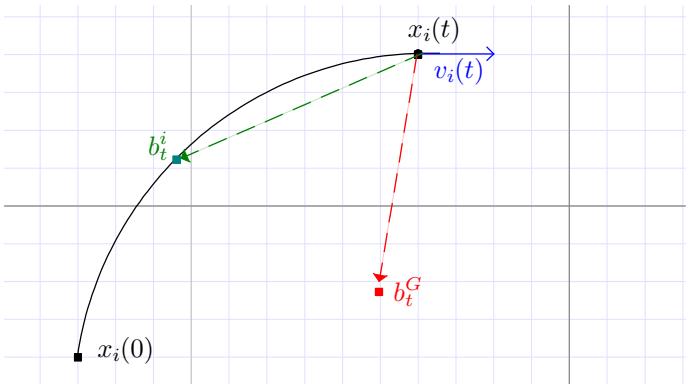

Figure 3. A simple representation of the **PSO**

To better understand the **PSO** and check that my implementation is correctly working, I chose to test it on a simple 2D problem, the (inverted) fitness landscape shown in figure 4.

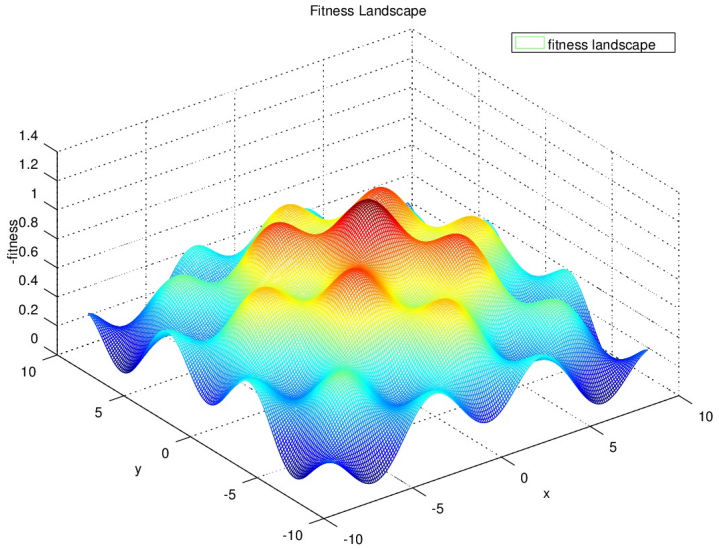

Figure 4. simple 2D landscape

Running the code with a population of 10 particles is getting the optimal solution in less of 20 iterations (and 0.1 ms), the execution is shown in figure 5.

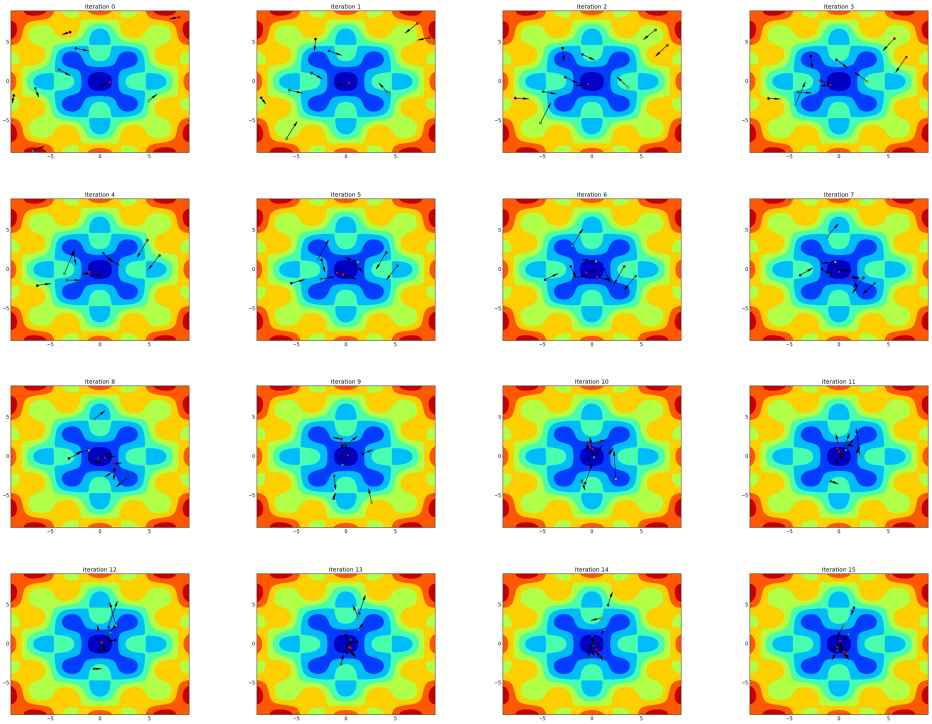

Figure 5. Evolution of the **PSO** in a simple landscape

As can be seen, the particles converge quite fast in the optimal region, but for a such reason is possible that ,in a more rough landscape, the particles will converge in a sub-optimal region, as shown in figure 6 for a maximisation problem.

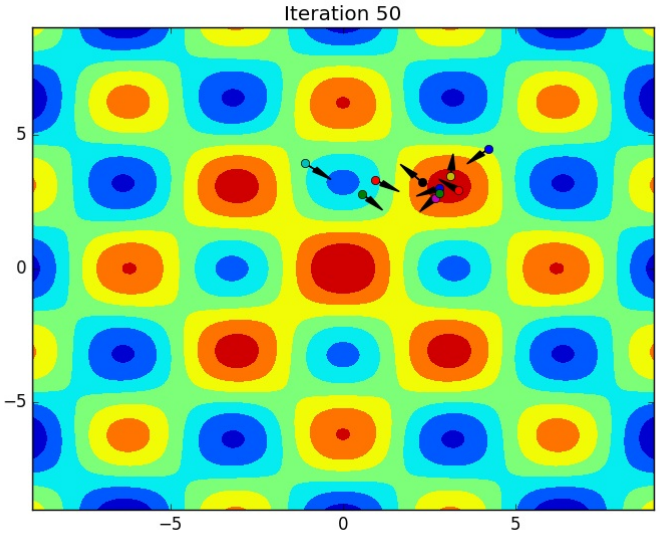

Figure 6. Non optimal results in a more rough landscape

Only an optimal configuration of ω, c_1, c_2 and m (the number of particles) values can improve the performance of the **PSO** in rough landscapes.

3 A NN to classify hand-written digits

In this TP we will specialise a **NN** to classify hand-written digits, in particular to distinguish between the digits "2" and "3".

The training set is composed by 200 digits, represented as 20×20 pixel gray-scale images, with their relative label (1 for "2" and 0 for "3").

The **NN** will be composed by 3 layers, the input layer will have one neuron per pixel (400), to download the image, and one more neuron for bias. The hidden layer will be composed of 25 neuron, plus one of bias. And finally the output layer will contain only one neuron (as the label is one value only).

θ_1 is a matrix of size 401×25 and represents the arcs between the input layer and the hidden one. θ_2 is a matrix of size 26×1 and represents the arcs between the hidden layer and the output.

The result of the transformation can be expressed mathematically as:

$$h_\theta(x) = g([1; g([1; x] \times \theta_1)] \times \theta_2)$$

beeing $g(x)$ is the sigmoid function.

The total number of parameters to optimise is 10051.

4 PSO and our NN

As for any other metaheursitic we have to define some elements:

- the solution space S
- a start solution x_0
- the neighbourhood of a solution $N(x)$
- the exploration operator U
- a fitness function
- the stop condition

The solution space S for training our **NN** is $S \subset \mathbb{R}^{10051}$ (where 10051 is the number of the parameter to optimise, as explained before), and I chose to limit each dimension between -5 and $+5$.

The start solution of each particle of the **PSO** will be a random position $x_i(0) \in S$.

The neighborhood of a solution $N(x_i(t))$ is composed by only one element computed by the exploration operator U as explained before in section 2.

The fitness function $J_k(\Theta_1, \Theta_2)$, already introduced, will be the mean squared error of the transformation $J(\theta_1, \theta_2) = \frac{1}{n} \cdot \sum_{k=1}^{n} \|h_\theta(x_k) - y_k\|^2$.

The stop condition is a total number $t_{\max}$ of iterations to reach.

5 Implementation

I implemented the algorithm in C++ extensively using the object-oriented features of the lanuage, and reusing all the metaheurstic framework developed till now. I added few classes to represent solutions in continuos spaces. First I chose to implement by myself a simple matrix and vector utility (*matrix.hpp* and *matrix.cpp*), but finally I opted to use a faster and existing open source library, called *eigen*, that makes use of multi-threading and vectorization to have better performance.

Since I tried to generalize as much as possible all the code, was very simple to write the samples code (which results are shown before in figure 5 and 6), for both the **PSO** and the **NN**.

In particular to optimize the performance, I chose to apply the **NN** transformation to the whole training set in a single operation.

Moreover the code implements a cut-off velocity $v_{\max}$ and a simple border bounce method.

6 Results

After the simple tests shown before, I chose to make a simple random analysis of the **NN** landscape by extracting 1000 random solutions from S. The result is shown in figure 7.

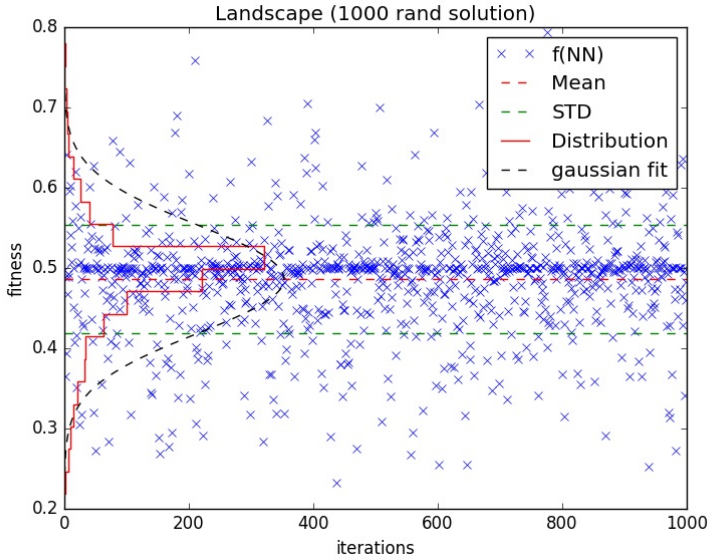

Figure 7. Sampled landspace of the **NN** problem

The fitness function will always return a value between 0 and 1, where 0 is the ideal and optimal case and 1 is the worst case. Clearly in figure 7, the value 0.5 is predominant, indeed the mean is 0.48 while the standard deviation is only 0.07. Finally the best fitness discovered with a random sampling is 0.22.

To choose the optimal parameters of the **PSO**, I wrote a small *bash* script to vary the parameters and generate a *csv* file with the results, plotted in figure 8.

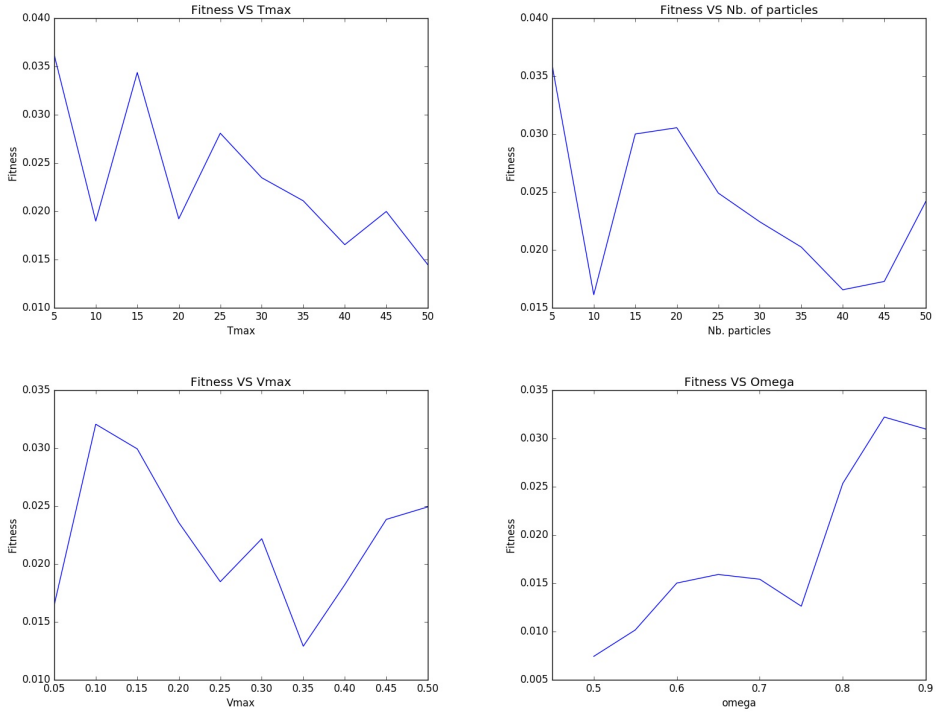

Figure 8. Fitness VS parameters $t_{\max}, m, v_{\max}, \omega$

The results of this analysis are interesting: on one side, as expected, higher $t_{\max}$ and m (number of particles) give better results, on the other side higher values of ω deteriorate the performance as well as too slow or too fast $v_{\max}$ values.

With higher number of particles, the **PSO** explores more precisely more parts of S.

Particles with too slow $v_{\max}$ will need too much time to explore S, while with too fast $v_{\max}$ the particles will not be precise enough (they will "jump" optimal regions).

For this reasons, I chose the following parameters:

$t_{\max}$	m	ω	c_1	c_2	$v_{\max}$
80	40	0.75	1.8	1.2	0.35

Finally running the code 50 times gave me the results shown in figure 9

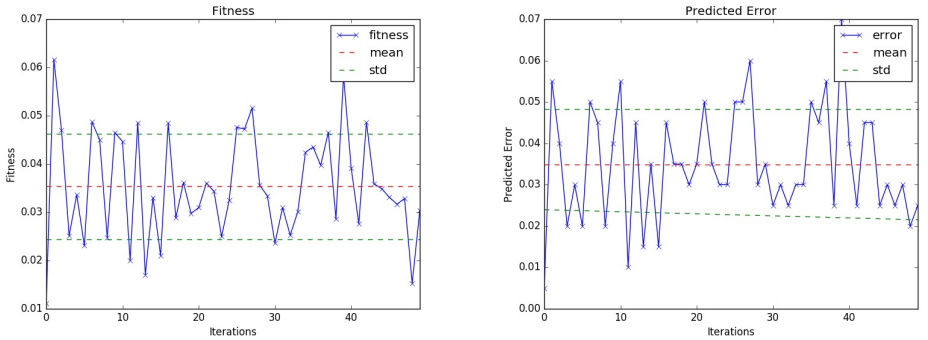

Figure 9. Fitness and Predicted Error of 50 runs

The fitness and the predicted error are strictly correlated as it is possible to see in figure 9. The performance of the optimized **NN** are very impressive, with in the best case a predicted error of only 0.005 (so a precision of 99.5%). Moreover the **PSO** needs only 3.1 seconds to train the **NN**. Personally I found the combination of **NN** and **PSO** very intersting both in quality and computational performances. The following table summerizes the results:

	Mean	STD	Best Results
Fitness	0.035	0.011	0.011
Predicted Error	0.035	0.013	0.005

To conclude in figure 10 is shown the evolution of 20 particles over 40 iterations with the parameters discussed previously. A

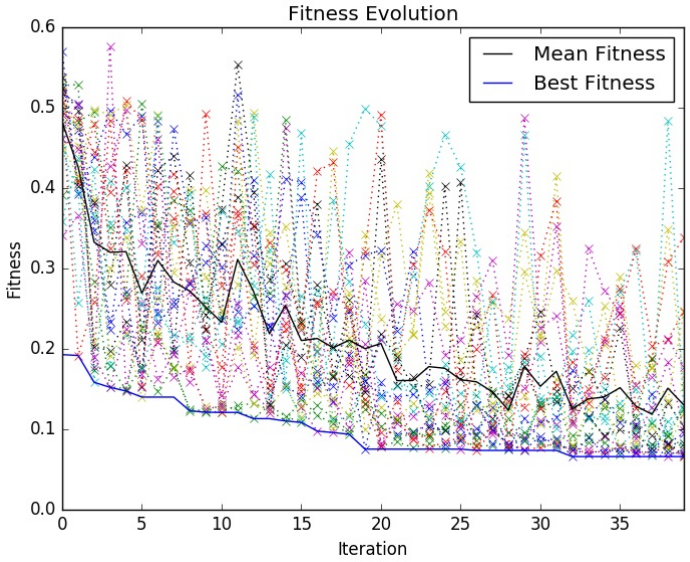

Figure 10. Evolution of the fitness in time

In this plot is possible to see how the particles converge while exploring the optimal region. In particular the trend of the mean fitness of all particles is very similar to the trend of the group best fitness. The dotted plots represent the fitness of every single particle. This result is some how similar to the result found for simpler problem, shown in figure 5.

7 Project structure

compile.sh	is the compilation script
src/	contains all the source codes (above the core codes)
→/rsolution.*	is the code that defines a solution $\in\mathbb{R}^n$
→/pso.*	is the code that implements the **PSO** metaheuristic
→/nn.*	is a generic implementation of the **NN**
→/mynn.*	is the extension of the **NN** for our test case
bin/	contains all the binary, dat files and the compilation script
→/steps	runs step by step (saving the status) the **PSO** over a simple problem
→/simple_pso	runs the **PSO** over a simple problem
→/pso	trains the **NN** using the **PSO**
→/steps_nn	runs the **PSO** on the **NN** problem step by step
→/multi_try.sh	script used to run multiple times the *pso* for statistics
→/variators.sh	script used to analyse the influence of the parameters over the fitness
bin/results	contains the csv generated and other output files

All the executables when executed without arguments will show the basic usage.

The codes and the reports can also be found on my github repository: Metaheuristic (branch eigen3).