Exercise 2: Swarm Intelligence

Haubenburger Gabriel (11840531) Seka David (11902064)

2024-12-19

1 Abstract

In this assignment, we aim to gain a experience in Particle Swarm Optimization. To that end, we optimize classic continuous functions with a tool based on NetLogo to gain intuition. Then, we apply our results to find a new way to optimize Neuronal Networks, a crucial task in Artificial Intelligence. The key issue in this regard is the tuning of the hyperparameters.

2 Implementation

Our implementation of the fitness function for the PSO-NN is rather straightforward. In a first attempt, we simply passed each of the solution points from PSO to the forward_prop() method of the neural network, and returned the array of calculated losses. We then noticed that NaN values can occur as a loss value, which can cause the framework to crash. To deal with this case, in our second attempt we filter out such values and replace them with a sufficiently large numerical value (100000), so solutions containing NaN values would not get picked.

For tuning the parameters, we first adjusted the parameters n_hidden, activation, n_iteration and learning_rate in commonsetup.py for each dataset. Afterwards, we further tuned the PSO-NN implementation in PSO-NN.py, in order to attempt to surpass or at least match the classic NN implementation. More details on this can be found in the next section.

3 Experiments

3.1 NetLogo Experimentation

We downlodaded NetLogo 6.4 and the tool "Netlogo_PSO_Playground" provided in Tuwel. We added an export function to this framework to be able to export the settings and findings of our experiments. We do the experiments with the Shubert function.

Unless stated otherwise, we use a swarm size of 40 an inertia of 0.9, a personal confidence of 1.0, a swarm confidence of 0.5, a speed limit of 10 and no constraint. We freeze the rest of the parameters to be able to compare the different tests with each other.

Due to the fact that the experiments are done on a GUI by hand, the number of datapoints we can feasibly find is naturally limited.

Fortunately, with around 10 experiments, one can see a clear tendency in most experiments.

3.2 PSO-NN Optimization

3.2.1 Iris Data Set

For the classic-NN, we started with simple a grid search over the activation functions (from SIGMOID, TANH, RELU, LEAKY_RELU, ELU, SOFTMAX, SWISH, SOFTPLUS, GELU) and the learning rate (from discrete values 10^n, $n = -6..1$). The activation function SIGMOID together with a learning rate of 0.1 gave us perfect accuracy of 1.0 in the given test setup in 10 experiments. The number of neurons in the hidden layer and the number of iterations were kept at 10 and 1000, respectively.

For the PSO-NN, we carried over the relevant optimized values from the classic-NN and also did simple a grid search over c_1, c_2 (each from the discrete values $i = 0.1..1.9$) and ω (from discrete values $n = -0.1..0.9$). By setting ω to 0.9 from the default 0.1 and c_2 to 0.3 was enough to achieve perfect accuracy of 1.0 in the given test setup in 10 experiments. c_1, swarm size and batch size were kept at the initial $0.1, 100$ and 200, respectively.

We used an early stop for the grid searches in order to stop as soon as the accuracy hit 1.0.

3.2.2 Breast Cancer data set

For the classic-NN, we once again started with simple a grid search over the activation functions (from SIGMOID, TANH, RELU, LEAKY_RELU, ELU, SOFTMAX, SWISH, SOFTPLUS, GELU) and the learning rate (from discrete values 10^n, $n = -6..1$). The activation function GELU together with a learning rate of 0.001 gave us an average accuracy of 0.94 in the given test setup in 10 experiments. The number of neurons in the hidden layer and the number of iterations were kept at 10 and 1000, respectively.

We tried again with increasing the number of neurons in the hidden layer to 15, then 20. In both cases, the average accuracy capped at 0.94 as well. Only by increasing the neurons in the hidden layer to 30 did we manage to achieve an average accuracy of .95 with the activation function LEAKY_RELU and a learning rate of 0.01.

For the PSO-NN, we carried over the relevant optimized values from the classic-NN and also did simple a grid search over c_1, c_2 (each from the discrete values $i = 0.1..1.9$) and ω (from discrete values $n = -0.1..0.9$). By setting ω to 0.9 from the default 0.1 and c_2 to 0.2 was enough to achieve an average accuracy of .96 in the given test setup in 10 experiments. c_1, swarm size and batch size were kept at the initial $0.1, 100$ and 200, respectively.

This time, we used an early stop for the grid searches in order to stop as soon as the accuracy hit 0.95 due to slower convergence.

3.2.3 Glass data set

For the classic-NN, we once again started with simple a grid search over the activation functions (from SIGMOID, TANH, RELU, LEAKY_RELU, ELU, SOFTMAX, SWISH, SOFTPLUS, GELU) and the learning rate (from discrete values 10^n, $n = -6..1$). The activation function SWISH together with a learning rate of 0.01 gave us an average accuracy of 0.94 in the given test setup in 10 experiments. We also needed to increase the number of neurons in the hidden layer and the number of iterations to 50 and 1500, respectively.

Staying at 1000 iterations, the best average accuracy we achived was .85 for 100 hidden neurons with the LEAKY_RELU activation function and a learning rate of 0.1.

For the PSO-NN, we carried over the relevant optimized values from the classic-NN and also did simple a grid search over c_1, c_2 (each from the discrete values $i = 0.1..1.9$) and ω (from discrete values $n = -0.1..0.9$). Due to the search taking much longer from more neurons and more iterations, we interrupted this grid search to manually try various values, mainly for c_1 and c_2 (for ω we went with a high value, since these seemed to work better in other data sets). By setting ω to 0.9 from the default 0.1 and c_2 and c_2 both to 0.8 was enough to achieve an average accuracy of .98 in the given test setup in 10 experiments. Swarm size and batch size were kept at the initial 100 and 200, respectively.

4 Results and Analysis

4.1 NetLogo Results

First of all, we test how different swarm sizes impact the convergence behaviour.

pop size	Try 1	2	3	4	5	6	7	8	9	10
10	–	27	–	16	27	39	34	–	38	31
20	–	–	11	37	40	43	30	32	–	29
40	6	18	22	26	22	21	24	29	13	25

Tabelle 1: Runs of varying population sizes. Data: number of iterations if converged, otherwise –

As we see in Table 1, a population size of 40 seems is enough to avoid starvation nearly all the time, and reduces the iterations needed to find the optimal solution. However, there seems not too much difference between a population size of 10 and 20.

Next up, we test different inertia values. As 40 seems to be a good value for the population size, we stick with it and leave the other parameters unchanged.

In Table 2, we see that for our setup, a wide range of inertias is possible without a big impact on the chance of the model to find a solution. However, one notices a trend that very high or very low values might lead to a longer time to convergence.

We continue with different confidence ratios. To that end, we first fix all the parameters but the personal confidence (the swarm confidence stays at 0.5) and report the impacts. Afterwards, we fix all the parameters but the swarm confidence (the personal confidence stays at 1) and report the impacts.

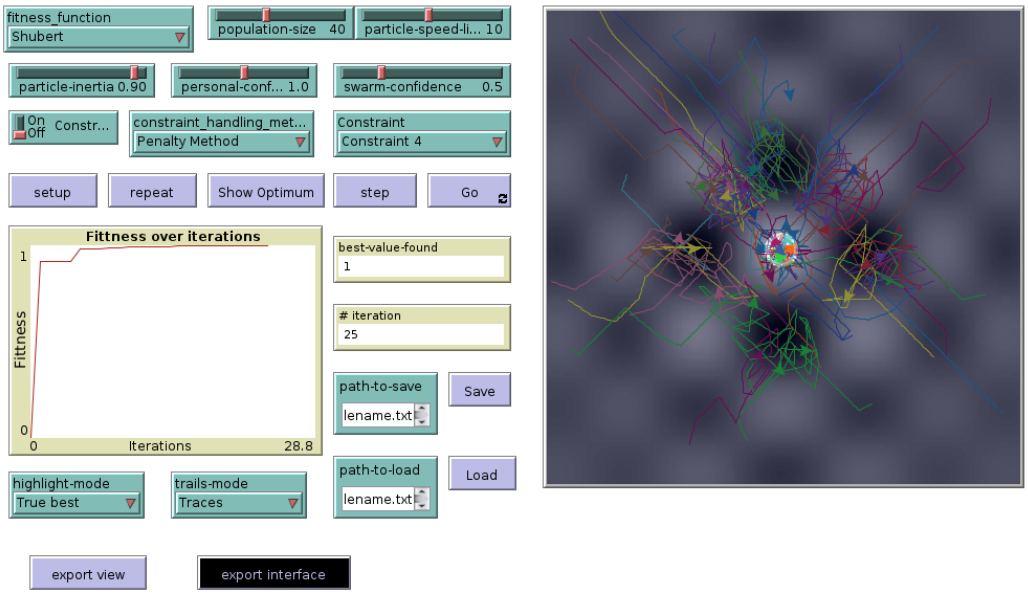

Abbildung 1: The setup for the population size test.

inertia	Try 1	2	3	4	5	6	7	8	9	10
.99	14	31	25	370	37	28	23	24	61	22
.7	26	12	20	26	16	13	13	20	14	13
.15	17	20	39	54	22	13	73	83	37	25
.05	55	182	42	23	–	75	46	109	15	28

Tabelle 2: Runs of varying inertias. Data: number of iterations if converged, otherwise –

personal confidence	Try 1	2	3	4	5	6	7	8	9	10
0	25	32	36	25	29	21	40	39	31	24
0.5	8	12	17	–	14	6	19	13	24	21

Tabelle 3: Runs of varying personal confidence. Data: number of iterations if converged, otherwise –

swarm confidence	Try 1	2	3	4	5	6	7	8	9	10
0	–	–	–	–	–	–	–	–	–	–
0.1	30	6966	42	32	6006	19	28	43	34	60
0.2	182	51	40	53	28	26	41	22	40	1523

Tabelle 4: Runs of varying personal confidence. Data: number of iterations if converged, otherwise –

For our problem, the personal confidence seems to have a rather small impact on the convergence speed, as we can see in Table 4. This can be due to the fact that the swarm is already big enough to take care of the exploration with the initial assignment. Having no swarm confidence has a big impact, as we can see in Table 5. Our explaination for this phenomenon is that the global best value is the only thing that pulls particles out of

local optima. Not having a sufficiently high pull-factor puts the burden of improving the global best value to the few particles that get pulled towards it, as we can see in figure 2.

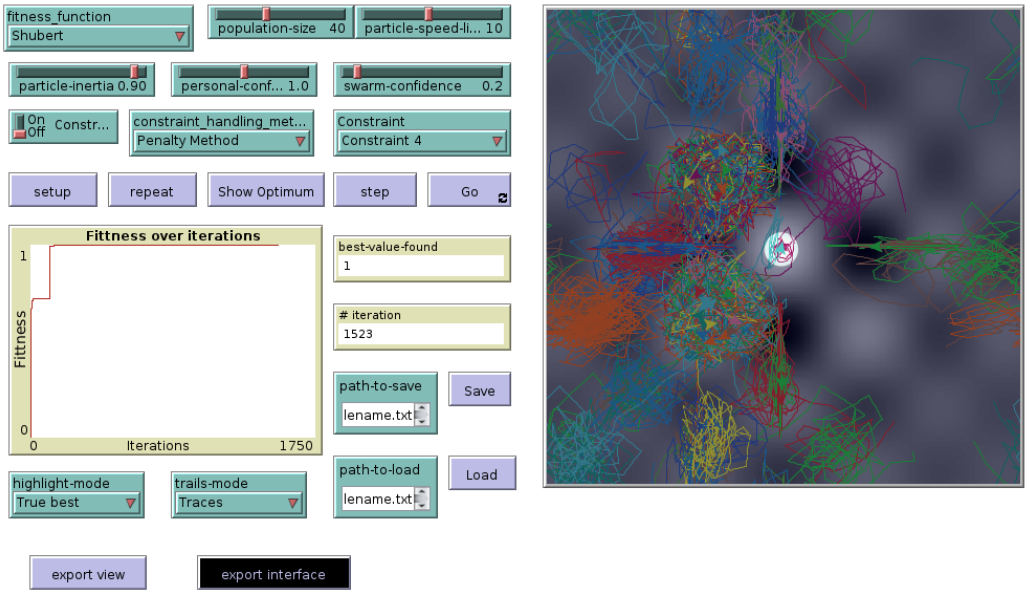

Abbildung 2: A run with a low swarm confidence.

Another key parameter to tune is the maximum velocity. As this parameter does not seem to have a big influence whether the PSO converges or not, we present some figures of different max velocities. We want to highlight the fact, that the trajectories with he lower speeds go straight towards the global optimum (best seen in figure 4) until they get stuck in a local minimum (figure 3). For larger maximum speeds, the particles tend to jump around erratically (figures 5,6)

This leads us to the assumption, that higher maximum speeds benefit exploration as particles can pass local maxima. On the other hand, lower maximum speeds benefit exploitation, as particles stay in the neighborhood of promising values.

4.2 PSO-NN Optimization

For the first two data sets, the classic NN was able to achieve decent results with only automatic parameter tuning in a reasonable time frame. Here PSO was only at best able to match the classic NN with the approach to parameter tuning.

For the Glass data set on the other hand, picking parameters by hand allowed as to surpass the classic NN by a bit. Automatic parameter tuning for the PSO-NN was infeasible at this classic-NN size however.

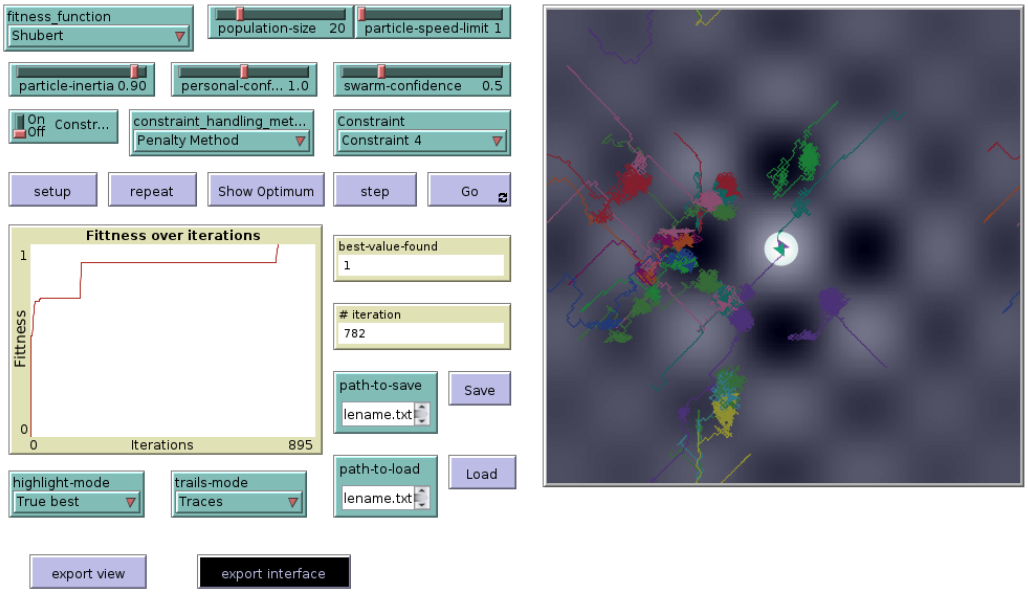

Abbildung 3: A run with a max speed of 1.

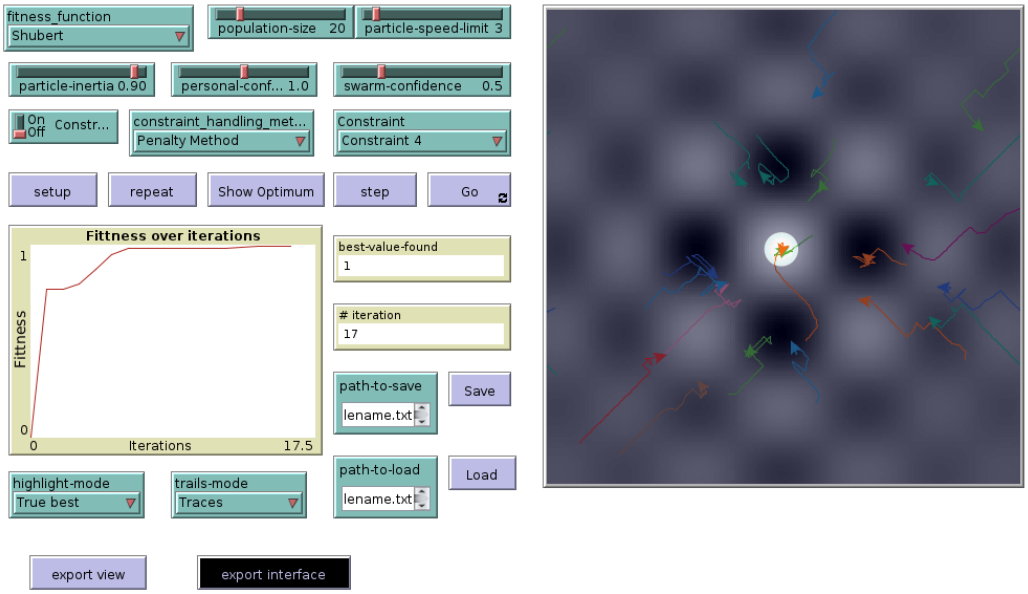

Abbildung 4: A run with a max speed of 3.

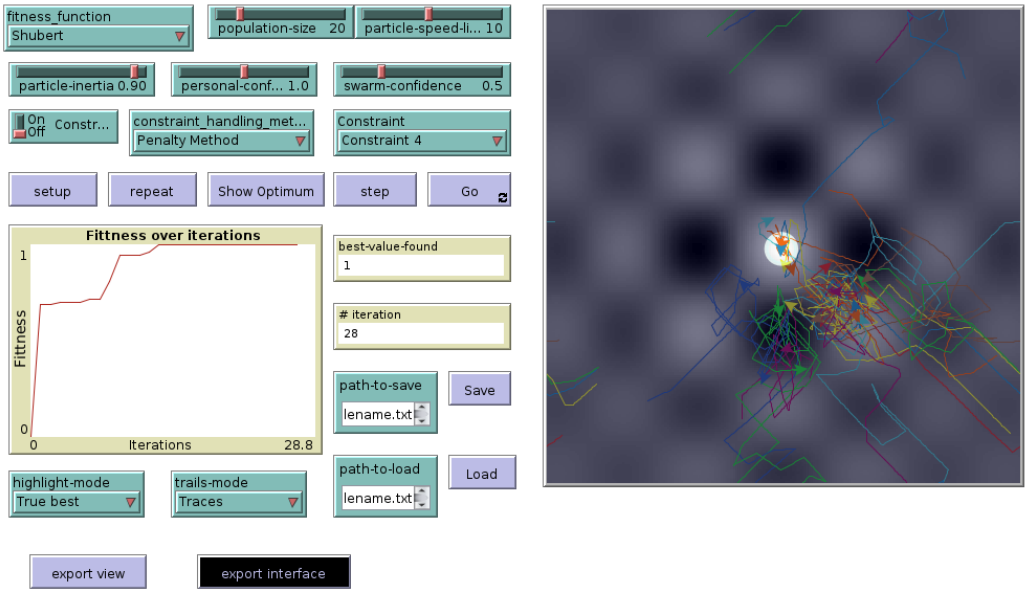

Abbildung 5: A run with a max speed of 10.

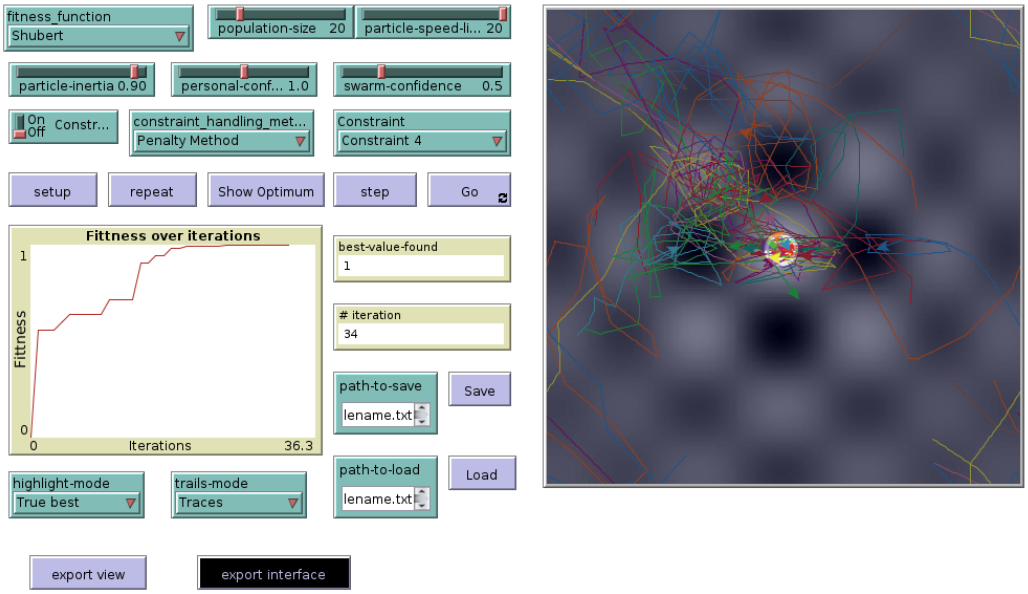

Abbildung 6: A run with a max speed of 20.

5 Conclusion

We can conclude that for small neuronal networks, PSO can be a potent alternative to back propagation. Its strength is that it can avoid the local minima the usual back propagation is designed to tend towards. Its weakness is that it requires elaborate hyper-parameter optimization. While with back propagation, one only has to tune the learning rate, for PSO one needs to fine-tune c_1, c_2, w as well as the size of the swarm.