Project #1

Optimization Methods for Machine Learning - Fall 2022

Nicolò Bertini
Business intelligence and analytics
bertini.1844162@studenti.uniroma1.it

Leonardo Chichiriccò
Modelli decisionali per l'ingegneria gestionale
chichiriccò.1853964@studenti.uniroma1.it

Fabio Ciccarelli
Modelli decisionali per l'ingegneria gestionale
ciccarelli.1835348@studenti.uniroma1.it

***Abstract*—In this paper we are going to discuss and analyze the implementation of different model of Neural Networks based on multilayer perceptron and radial basis function.**

I. Shallow MLP network implementation

A. Defining cost and gradient functions:

The *cost function* used is:

$$E(w,\pi) = \frac{1}{2P}\sum_{p=1}^{P}\left(f(x^p) - y^p\right)^2 + \frac{\rho}{2}\|w\|^2$$

Whose *gradient* can be computed in a closed form as follows:

$$\nabla E = \begin{bmatrix} \nabla_w E \\ \nabla_v E \end{bmatrix} + \rho \begin{bmatrix} vec(W) \\ v \end{bmatrix}$$

$$where \;\; \nabla_w E = \Big[[(v^T \otimes Z) \odot (G' \otimes \mathbb{1}_{1\times(n+1)})] \odot [\mathbb{1}_{1\times N(n+1)} \otimes (\hat{y} - y_t)]\Big]^T \mathbb{1}_{P\times 1}$$

$$\nabla_v E = (G^T G v - G^T Y)$$

To go deeper into these formulas, see **Appendix A**.

B. Final setting and evaluation of the occurrence of overfitting and underfitting

In order to evaluate the behaviour of the model with different combinations of the hyper-parameters, we performed a grid search K-folds cross validation (KCV) with the following possible values for N, σ and ρ.

$K = 6$
$N \in [1, ...80]\ s.t.\ N = 2k,\ k \in \mathbb{N}$
$\sigma \in \{0.5, 1, 2, 4, 8, 15, 20, 30\}$
$\rho \in \{10^{-5}, 0.5 \cdot 10^{-4}, 10^{-4}, 0.5 \cdot 10^{-3}, 10^{-3}\}$

We reported here the optimal parameters we obtained after the grid-search KCV:

	N	σ	ρ
Final parameters	62	1	10^{-5}

TABLE I
Optimal parameters for MLP

We then graphed the trend of train and validation scores for each hyper-parameter, averaging the errors obtained during the grid-search KCV fixing the parameter of interest and let the other one vary.

1) Number of neurons: As we can see from figure [1], when the number of neurons is set to a value lower than 5 the average error value, both on the training set and on the validation one, increases: this means that a similar MLP architecture is not enough complex to deal with the requested function approximation, and therefore we incur under-fitting. In such cases, we see a low score on both the training set and test/validation set data. On the other hand, as we can see, our model does not suffer of over-fitting problems for a number of neurons up to 80 (which is the maximum number of neurons we consider for the grid search, since we have not noticed any significant improvement increasing N).
Surely enough, further increasing the number of neurons over a certain tresh-hold would have led the model to over-fit, but we could not reach such values since in that case grid search would have lasted too much time.

2) σ: Concerning σ parameter, we notice that, increasing its value, we incur over-fitting. Note that, in figure [2], we plotted the error behaviour when fixing σ and averaging over all the possible combinations of the other parameters. In this case, there is just a slight over-fitting (in fact, training error worsen as well). On the other hand, when evaluating the errors in function of σ, fixing the other parameters to the optimal ones, we can notice a significant over-fitting on the upper values of the variable (see figure [3]).

3) ρ: In figure [4] we can see how the lower the parameter ρ, the less we penalize the complexity of the model.Therefore, the error on the training set decreases, while the one on the validation set increases, which is synonym of a slight over-fitting.

C. Building the optimization routine and analyzing its results

In order to get a local minimizer for the cost function *E*, we used the pre-built tool *scipy.optimize.minimize*, which takes as input the objective function, a starting point and some other optional arguments such as the Jacobian function or a tolerance on the gradient norm.
Since we built the Jacobian, and we wanted to impose a tolerance on the gradient norm equal to 10^{-4} as well (the one which gave us the most convenient trade-off between speed of convergence and optimal value of the objective function), the choice was restricted to a few possible optimizers, such as BFGS, Conjugate-Gradient and Newton-CG. The solver that guaranteed us the best performances in term of running time and succesfull optimization routine is BFGS.
The same reasoning has been followed for all the points of the project.

In the following table we report the final configuration of our model.

Final settings and results	
Solver	BFGS
Output message	Success
Starting value of the objective function	149.566
Final value of the objective function	0.001
Number of iterations	1062
Norm of the gradient at the starting point	94.7
Norm of the gradient at the final point	0.000246
Computational time	Up to 5s

TABLE II
Final settings and results

D. Computing the error's score and displaying the predicted function

In the table below final results are displayed.
Note that the results of the optimization routine are very sensitive to different starting points, since we have a highly non convex function with a lot of local minima, and therefore the starting point can lead the optimization procedure to be attracted from local minima very different from each other in terms of quality of the solution.

Training error	Test error
$8.11 \cdot 10^{-5}$	0.000594

TABLE III
FINAL RESULTS

In figure [5] the plot of the final function we obtain is shown.

II. SHALLOW RBF IMPLEMENTATION

A. Defining gradient function.

Also the gradient for the RBF network cost function can be computed in a closed form, as it follows:

$$\nabla E = \begin{bmatrix} \nabla_c E \\ \nabla_v E \end{bmatrix} + \rho \begin{bmatrix} Vec(c) \\ v \end{bmatrix}$$

$$where \quad \nabla E_c = \frac{2}{P\sigma^2}[((\hat{y} - y_{true}) \otimes \mathbb{1}_{Nn}^T) \odot A \odot (\Phi \otimes \mathbb{1}_n^T) \odot (v^T \otimes \mathbb{1}_{P \times n})]^T \mathbb{1}_P$$

$$\nabla_v E = \frac{1}{P}(\Phi^T \Phi v - \Phi^T y)$$

To go deeper into these formulas, see **Appendix A**.

B. Final setting and evaluation of the occurrence of over-fitting and under-fitting

In order to evaluate the behaviour of the model with different combinations of the hyper-parameters, for the RBF model as well, we performed a grid search K-folds cross validation with the following possible values for N (Number of centers), σ and ρ.

$K = 6$
$N \in [1, ...80]\ s.t.\ N = 2k,\ k \in \mathbb{N}$
$\sigma \in \{0.2, 0.5, 0.8, 1, 2, 5, 10\}$
$\rho \in \{10^{-5}, 0.5 \cdot 10^{-4}, 10^{-4}, 0.5 \cdot 10^{-3}, 10^{-3}\}$

In the following table we can see the optimal hyper-parameters:

	N	σ	ρ
Final parameters	52	1	10^{-5}

TABLE IV
OPTIMAL HYPER-PARAMETERS FOR RBF

We then graphed the trend of train and validation scores for each hyper-parameter, averaging the errors obtained during the grid-search KCV fixing the parameter of interest and let the other one vary.

1) Number of Centers: As we can see from the image [6], with a low number of centers the average error value, both on the training set and on the validation one increases: this means a RBF architecture with a low number of centers is not complex enough to deal with the requested function approximation, and therefore we incur under-fitting. We can also notice that our model doesn't run into over-fitting problems for an high number of centers up to 80, which is the maximum number of centers we considered for the grid search. As said before, we expect to have over-fitting increasing the number of centers up to the number of samples or above, since this could make the model interpolates the data.

2) σ: We can consider the RBF shallow network output as a local approximation of the true function, made through modulation and linear combination of a number N of gaussian functions having a fixed variance σ^2. Given a fixed number of centers N, performances of the network can significantly vary in function of σ value. In the case of a very small σ, in fact, we must have a huge number of centers, evenly distributed all over the domine of the true function, to achieve a good approximation of it. This comes from the fact that the local gaussian approximation drops to zero very quickly (since it has a very low variance), and therefore it can contribute only to approximate a tiny chunk of the true function. In the case in which we don't have a sufficient amount of centers, the model might have poor performances and, therefore, run into under-fitting. The same can happen if the gaussian functions have a very high variance (with respect to the size of the domain), since a change in the weight associated with one of them will affect the behavior of the predictor in a significant part of the domain, leading to a very complicated optimization procedure and, in most cases, to under-fitting.
This can be also noticed from figure [7], which represents the average error (both on train and validation set) made by our model depending on σ value.

3) ρ: In figure [8] we can notice a slight divergence of the error's score on the train test and the validation test for low values of ρ. Instead, for high values of ρ we can notice an increase of the average error's score, symptom of a slight under-fitting.

C. Building the optimization routine and analyzing its results.

In order to get a local minimizer for the cost function *E*, we can consider valid the same reasoning we did for the MLP implementation, reported in the previous section.
In the following table we report the final configuration of our model.

Final settings and results	
Solver	BFGS
Output message	Success
Starting value of the objective function	19.6
Final value of the objective function	0.000119
Number of iterations	526
Norm of the gradient at the starting point	10.37
Norm of the gradient at the final point	0.000298
Computational time	Up to 3s

TABLE V
FINAL SETTINGS AND RESULTS

D. Computing the error's score and displaying the predicted function

In the table below we displayed the final results.

Training error	Test error
0.000174	0.000362

TABLE VI
FINAL RESULTS

In figure [9] the plot of the final function we obtain is shown.

III. EXTREME LEARNING

A. Extreme learning for MLP

Extreme learning (EL) consists in making the full optimization procedure easier by fixing randomly the weights of the first layer and therefore

solving the convex problem represented by the tuning of weights v. In our case, we perform the routine described above for a certain number of iterations (we decided to fix the maximum number of iterations to 10.000), saving as the best solution the one which gives us the minimum value for *E*.
Note that each component of W is sampled from a uniform distribution in the interval [-2,2]. At each iteration, set the coefficients of W as said before, we then solve the following linear system:

$$(\frac{1}{P}G^TG + \rho I)v = \frac{1}{P}G^TY$$

Which corresponds to find a solution for $\nabla_{v|W}E = 0$ (which is the sufficient condition for optimality in case of a strictly convex optimization problems).
The method we used is *np.linalg.lstsq*, since it allows to solve the linear system in a very efficient way and in a short amount of time.
Note that we could have used a standard optimization routine as well, such as the one described in the previous sections, but in this case using a solver like BFGS or a Newton-based approach would have been way less efficient or not functioning at all. The comparison between the results of the EL and full optimization for MLP are reported in the table below.

	Train error	Test error
Full optimization	$8.11 \cdot 10^{-5}$	0.000594
Extreme learning	0.001-0.003	0.003-0.007

TABLE VII
COMPARISON OF THE FINAL RESULTS

As expected, results for extreme learning are worse than the ones obtained with the full optimization procedure. Such a decrease in terms of performance is due to the limited amount of iterations for the EL. In fact, if we have had the possibility to iterate over an infinite amount of randomly selected W, we would have achieved the optimal solution through this procedure as well.
Note that the fact that the results vary in a certain range is due to the random generation of the weights.

In figure [10] the comparison between the plot of the EL predicted function and the real function is shown.

B. Unsupervised selection of the centers for RBF

The unsupervised selection of the centers (USC) for RBF is the equivalent of the EL, applied to the choice of the coordinates of the N centers. In this case, we must randomly select N of the training samples as centers for the RBF, and then optimizing the coefficient given to each hidden neuron by solving the following convex problem:

$$(\frac{1}{P}\Phi^T\Phi + \rho I) = \frac{1}{P}G^TY$$

$$\nabla_v E = \frac{1}{P}(\Phi^T\Phi + \rho I)v - \frac{1}{P}G^TY$$

In this case as well the selection of centers is repeated a certain amount of times (we set the maximum number of iterations equal to 10.000).
For the same reasons explained before, also for this optimization routine we used the *np.linalg.lstq* method. The comparison between USC and full optimization for RBF network is reported in the table below.
As expected, results for extreme learning are worse than the ones obtained with the full optimization procedure. Such a decrease in terms of performance is due to the limited amount of iterations set for the USC.
In fact, if we have had the possibility to iterate over an infinite amount of randomly selected centers, we would have achieved the optimal solution through this procedure as well.
Note that the fact that the results vary in a certain range is due to the random selection of the centers.

	Train error	Test error
Full optimization	0.000174	0.000362
Extreme learning	0.0005-0.002	0.001-0.004

TABLE VIII
COMPARISON OF THE FINAL RESULTS

In figure [11] the plot of the comparison between the USC predicted function and the real function is shown.

IV. DECOMPOSITION METHOD

The decomposition method consists in dividing the main problem into two or more subproblems easier to solve. In this case, we apply a two blocks decomposition to the MLP network optimization problem already seen in I, dividing the selection of weights W (which represents a highly non-convex problem) from the tuning of the v values (which instead, as seen previously, is an easy to solve problem). To do this we can divide the procedure into two phases:

1) Tuning of v values: For the first phase we start with a random generation of the coefficients of the matrix W as done in III-A. Once fixed, we proceed to solve the quadratic problem associated with the linear system below:

$$\nabla_v E = \frac{1}{P}(G^TG + \rho I)v - \frac{1}{P}G^TY = 0$$

2) Finding the optimal W: Once the optimal weights v have been fixed as the output of the previous phase, we perform the minimization of the cost function only considering the W coefficients (with "BFGS" solver and *scipy.optimize.minimize* method). The minimization is carried out following the same procedure as the one used in I, where the gradient is only computed with respect to W:

$$\nabla_w E = \frac{1}{P}((V^T \otimes Z) \odot (G' \otimes \mathbb{1}_{1\times(n+1)}))\odot (\mathbb{1}_{1\times N(n+1)} \otimes (\hat{y} - y_t))^T \mathbb{1}_{P\times 1} + \rho \begin{bmatrix} \ vec(W) \ \end{bmatrix}$$

The two steps are repeated until the norm of the gradient drops below a certain threshold (in this case it is fixed to 0.0005) or the maximum number of iterations (equal to 1000) is reached . Also in this case results can vary in a certain range, this is due to the random generation of the starting point.
In fact, we decided not to fix the starting point for the optimization routine since this last method demonstrated to be more robust (with respect to the w_0 selection) than the previous ones.

The results of this routine are shown below:

Final settings and results	
N° of outer iterations	1 - 3
N° of sub-problems	2 - 6
N° of function evaluations	1500 - 3500
Computational time	10s - 20s

TABLE IX
FINAL SETTINGS AND RESULTS

The final values of the training and test error are reported here:

Training error	Test error
$5 \cdot 10^{-5} - 9 \cdot 10^{-5}$	$2 \cdot 10^{-4} - 8 \cdot 10^{-4}$

TABLE X
FINAL RESULTS

A. Comparison of results

This last optimization routine works better than the others considered so far. Moreover, it is way more robust than the other methods with respect to the starting point for the optimization, in the sense that it achieves almost every time very low error values (both on the training and on the test set) regardless of the w_0. This can be explained by the fact that, dividing the routine into two phases, it is unlikely for the optimization to be attracted by a poor local minimum. On the other hand, it requires slightly more time than the complete optimization procedure, but less than the Extreme Learning one (which is affected by the number of iterations chosen). Considering both these aspects, we think that the 2-blocks decomposition for MLP network is the best and most reliable optimization method analysed in this project, apart from the deep network proposed for the bonus question.

	EL for MLP	DM for MLP
Train error	0.0005-0.002	$5 \cdot 10^{-5} - 9 \cdot 10^{-5}$
Test error	0.000362	$2 \cdot 10^{-4} - 8 \cdot 10^{-4}$
N° of iterations	10000	1 - 3
Computational time	Up to 45s	10s-20s

In figure [12] the plot of the comparison between the DM predicted function and the real function is shown.

V. ADDITIONAL BONUS EXERCISE.

For the final exercise we constructed a deep neural network (DNN) with a variable number of layers *L* from scratch, computing the gradient of the error function via backpropagation. We evaluated different numbers of layers and neurons per layer and we got the best results (considering also the running time) using two hidden layers, each of 20 neurons. Note that the deep network we built is a MLP with the hyperbolic tangent as activation function and biases added to all the hidden layers. Moreover it is intended to be a the regression model, which means that $N_L = 1$. The hyperparameters σ and ρ have been set to the ones obtained via grid search in section [I]. In the following table it is reported the training error computed on all the 250 samples and in figure 13 the plot of the predicted function.

Training error	Test error
$2.14 \cdot 10^{-5}$	N.D.

TABLE XI
FINAL RESULTS

VI. FINAL COMPARISON

Comparing the overall performances of the different models, apart from the deep network, we notice the RBF model to have better results on average (both in terms of running time and error achieved). We do not know whether such an advantage is related only to this specific instance or it is valid for all the possible problems. Further analyses should be conducted.

A complete overview of the results of this project is shown below.

EX.		Settings		
		N	σ	ρ
Q 1.1	Full MLP	62	1	10^{-5}
Q 1.2	Full RBF	52	1	10^{-5}
Q 2.1	Extreme MLP	62	1	10^{-5}
Q 2.2	Unsupervised c RBF	52	1	10^{-5}
Q 3.0	Decomposition method	62	1	10^{-5}
Q 4.0	The Best Model	[20,20]	1	10^{-5}
EX.		Results		
		Final train error	Final test error	Optimization time
Q 1.1	Full MLP	$8.11 \cdot 10^{-5}$	0.000594	Up to 5s
Q 1.2	Full RBF	0.000174	0.000362	Up to 3s
Q 2.1	Extreme MLP	0.001-0.003	0.003-0.007	Up to 45s
Q 2.2	Unsupervised c RBF	0.0005-0.002	0.001-0.004	Up to 45s
Q 3.0	Decomposition method	$5 \cdot 10^{-5} - 9 \cdot 10^{-5}$	$2 \cdot 10^{-4} - 8 \cdot 10^{-4}$	10s - 20s
Q 4.0	The Best Model	$2.14 \cdot 10^{-5}$	N.D.	Up to 4m

VII. Figures and graphs

A. 1.1 MLP implementation

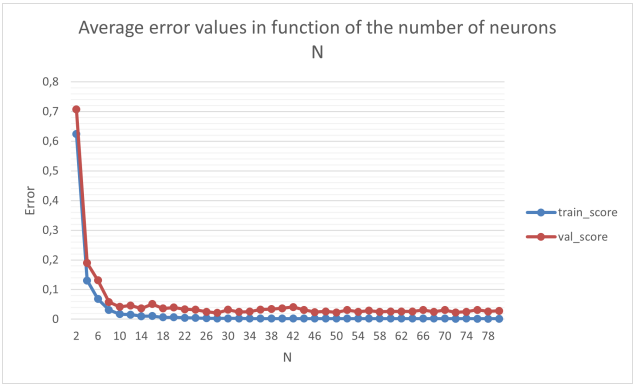

Fig. 1. Average error values in function of the number of neurons

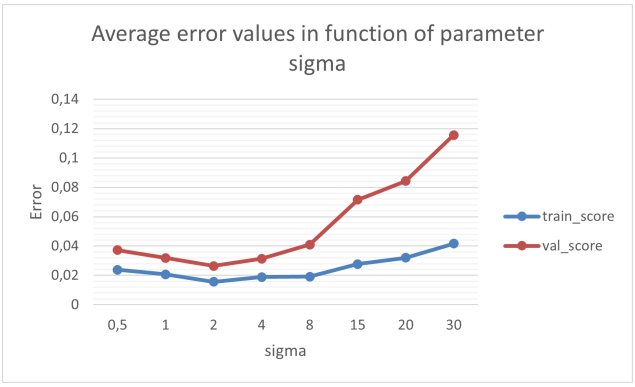

Fig. 2. Average error values in function of σ

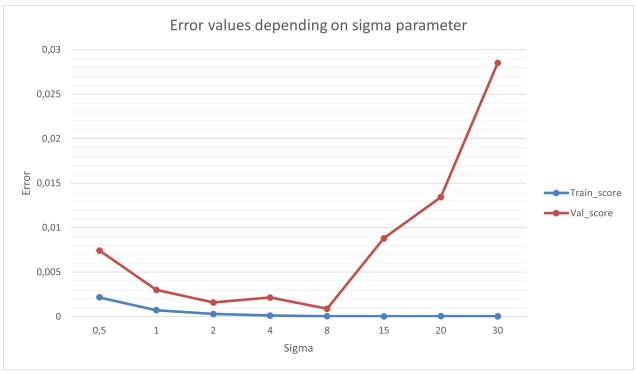

Fig. 3. Average error values in function of σ fixing the best parameters

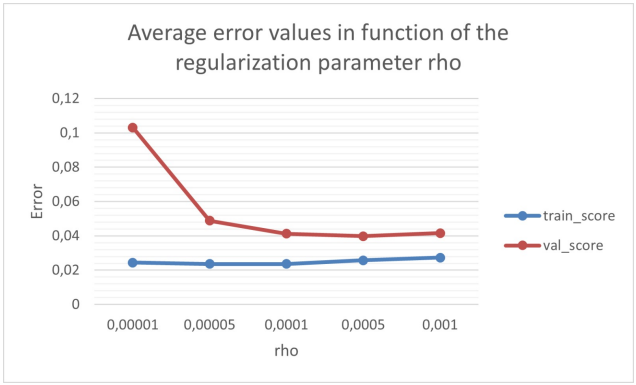

Fig. 4. Average error values in function of the ρ parameter

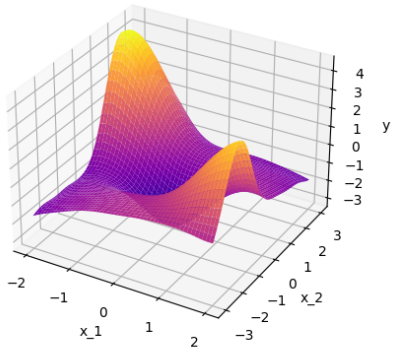

Fig. 5. Plot of the final function

B. 1.2 RBF implementation

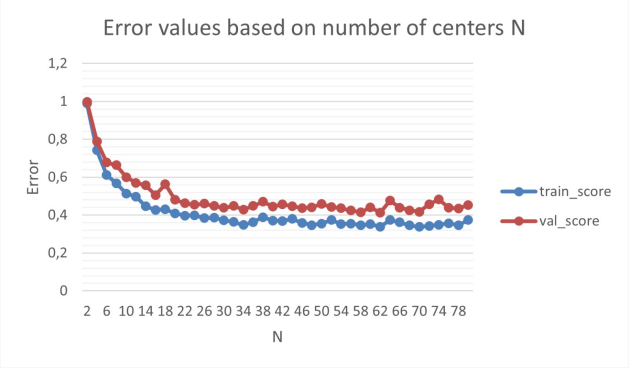

Fig. 6. Average error values in function of the number of centers N

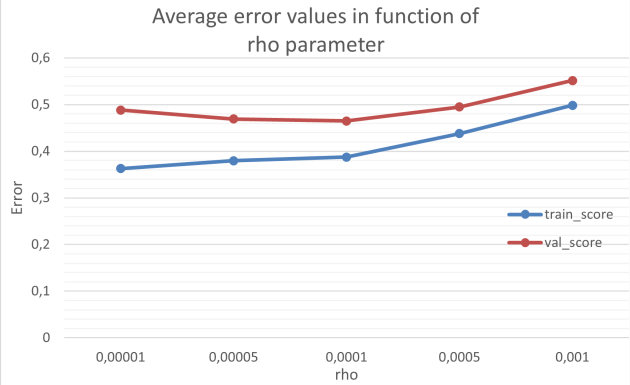

Fig. 8. Average error values in function of the ρ parameter

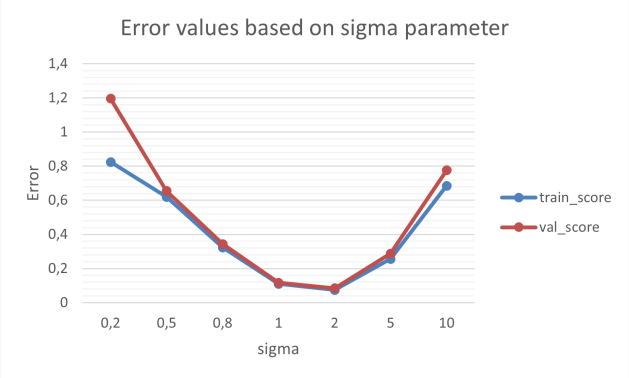

Fig. 7. Average error values in function of the σ parameter

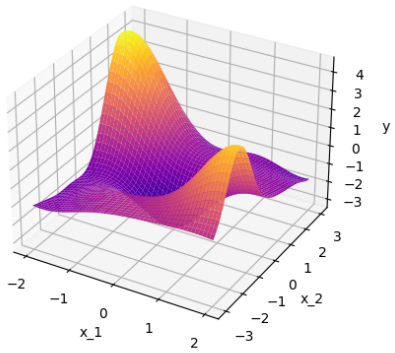

Fig. 9. Plot of the final function

C. 2.1 Extreme Learning MLP

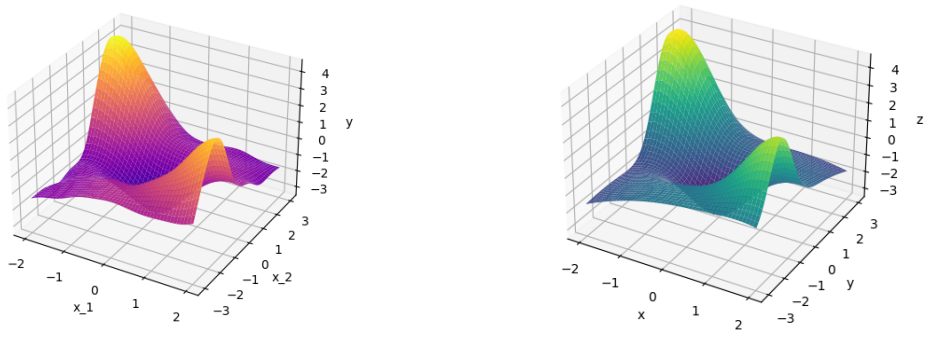

Fig. 10. Plot of the final function and the real one

D. 2.2 Extreme Learning RBF

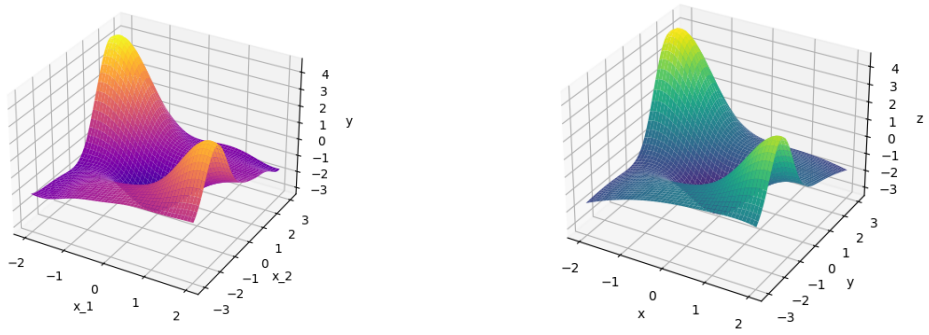

Fig. 11. Plot of the final function and the real one

E. 3. Decomposition method

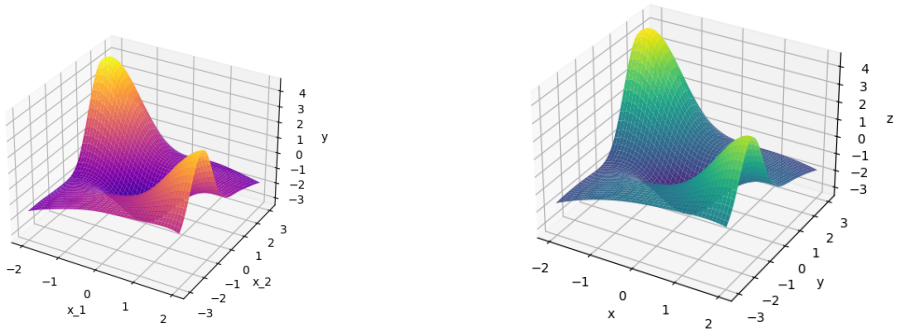

Fig. 12. Plot of the final function and the real one

F. 4. Best Model

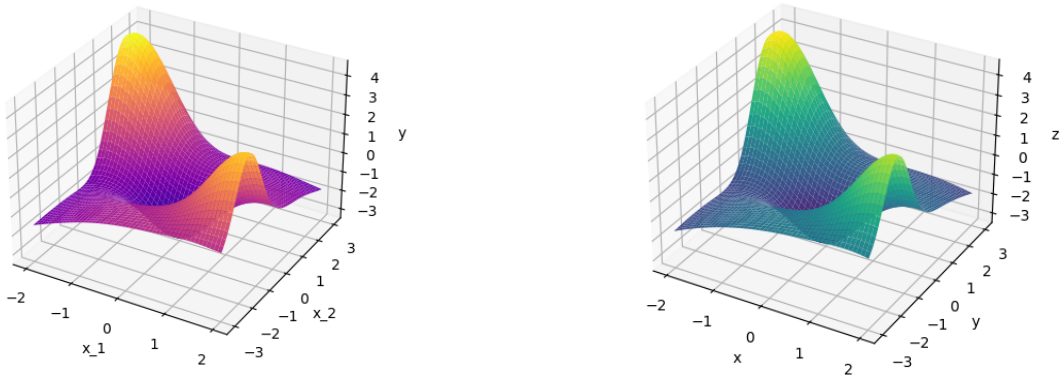

Fig. 13. Plot of the final Best Model function and the real one