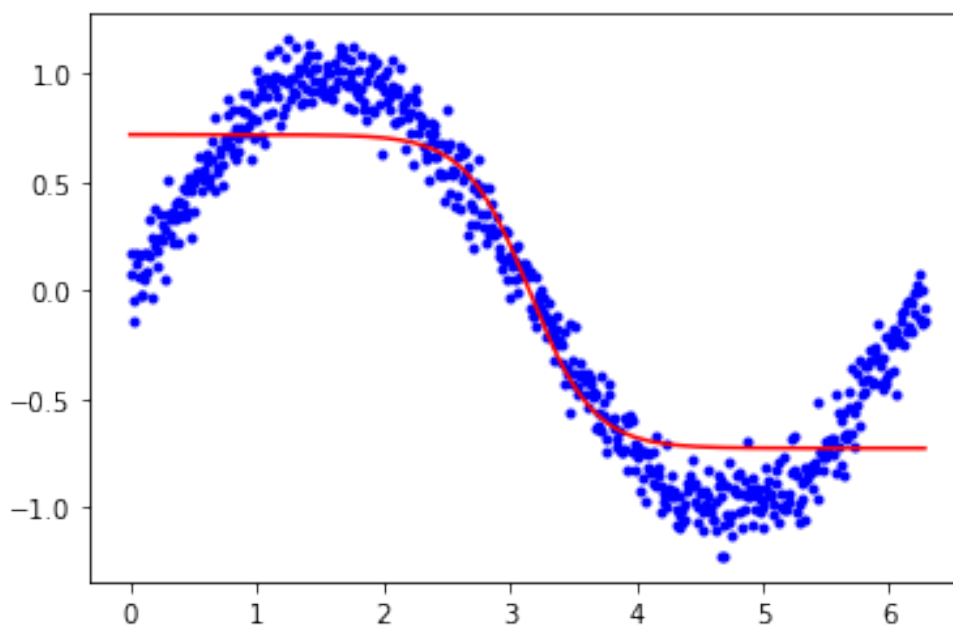


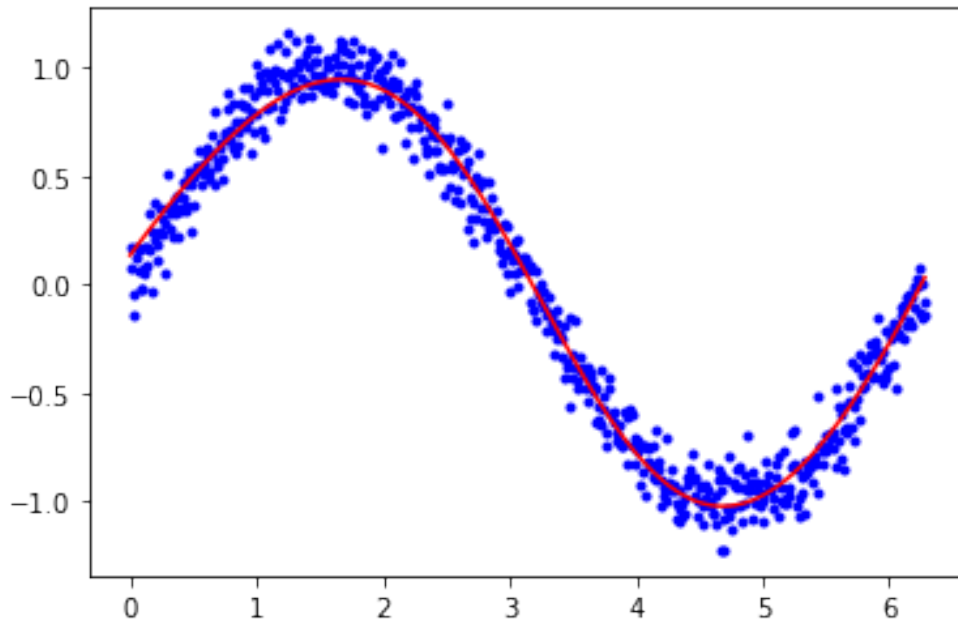
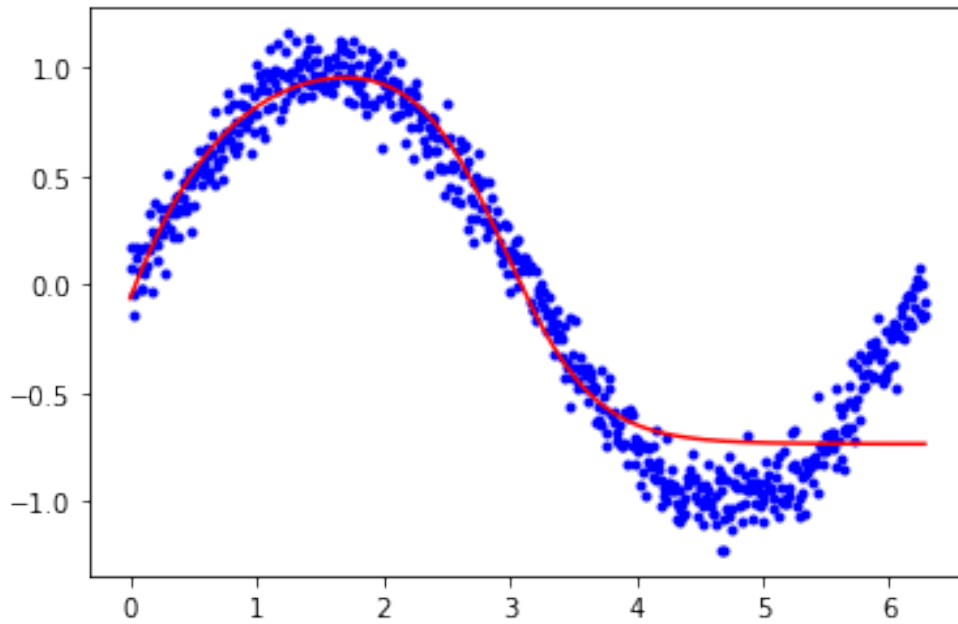
CHAP 8_Notebook N3 Multi-layer Perceptron for regression

1. Using 'tanh' and 'lbfgs' as parameters :

1. It is quite interesting to observe the case where a single hidden layer is used; for the activation function 'tanh', one observes the following regression result : One clearly recognizes the tanh function (up to the sign): As expected, the learner searches to fit the sine data by choosing parameters a, b, c and d such that the MSE between the data and $c \cdot \tanh(ax+b) + d$ is minimum.



For a single layer including 2 or 3 neurons, the result does not always lead to a « correct result » : 2 examples of obtained solutions are given below :



For 4 or more neurons in the single hidden layer, one gets similar good results, as can be confirmed by the MSE as a function of N_{neurons} ...

2. For a single hidden Layer counting N_{neurons} :

First consider the input layer and its connections to the hidden layer :

from entry x to each neuron = N_{Neurons} coeff

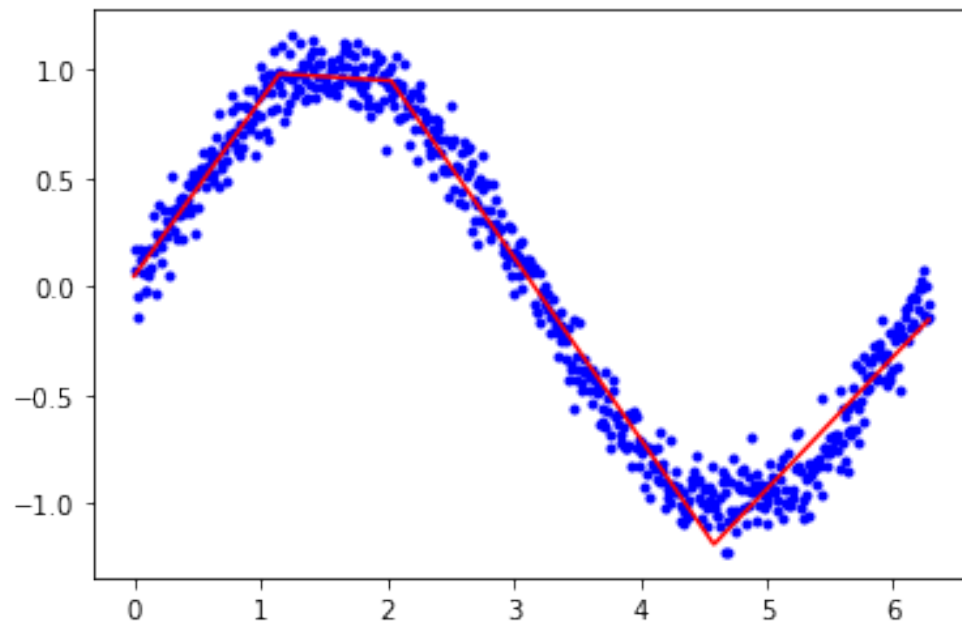
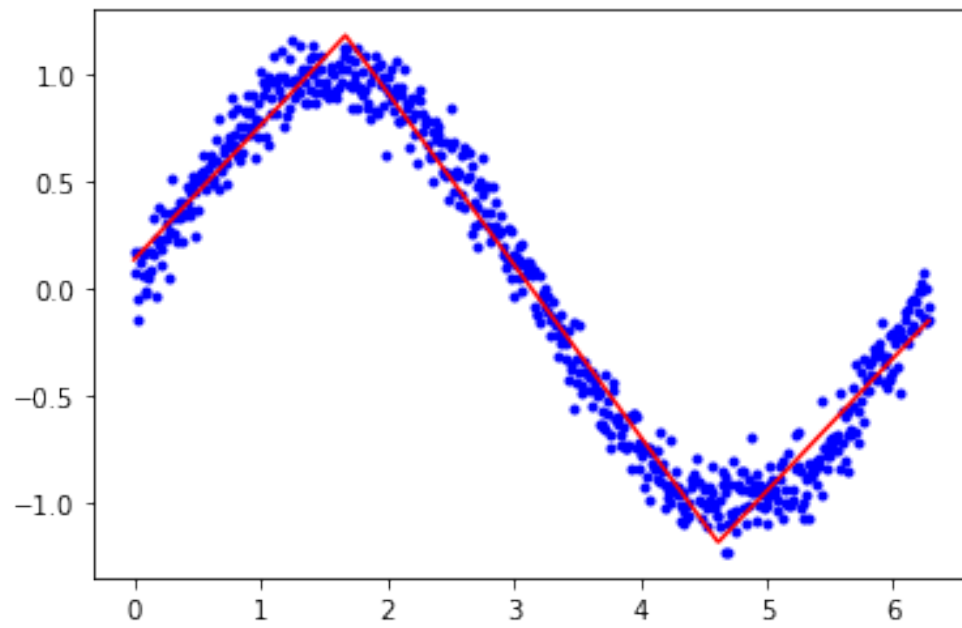
from bias (+1) to each neuron = N_{Neurons} coeff

Then from the hidden layer to the output layer : 25 coeff + 1 from bias = 26.

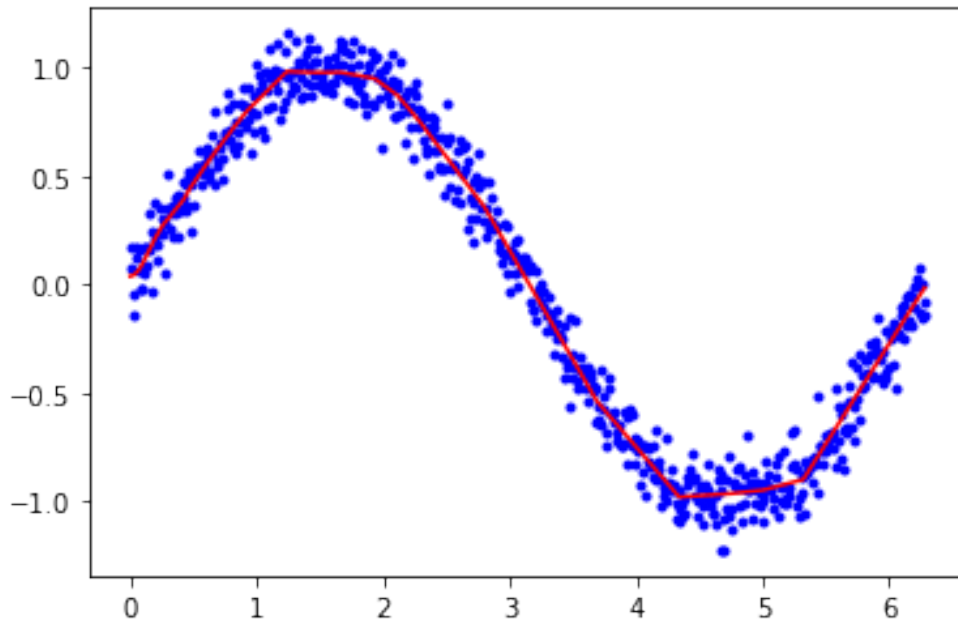
finally, this is a total number of coeff that is $3 \cdot N_{\text{neurons}} + 1$.

3. Replacing 'tanh' by rely in the activation function, and running the code for e.g.

$N_{\text{neurons}}=4$, the « best » results found look like



and below is an example of what we obtain for lbfgs solver (Newon like) and $N_{\text{neurons}}=100$:

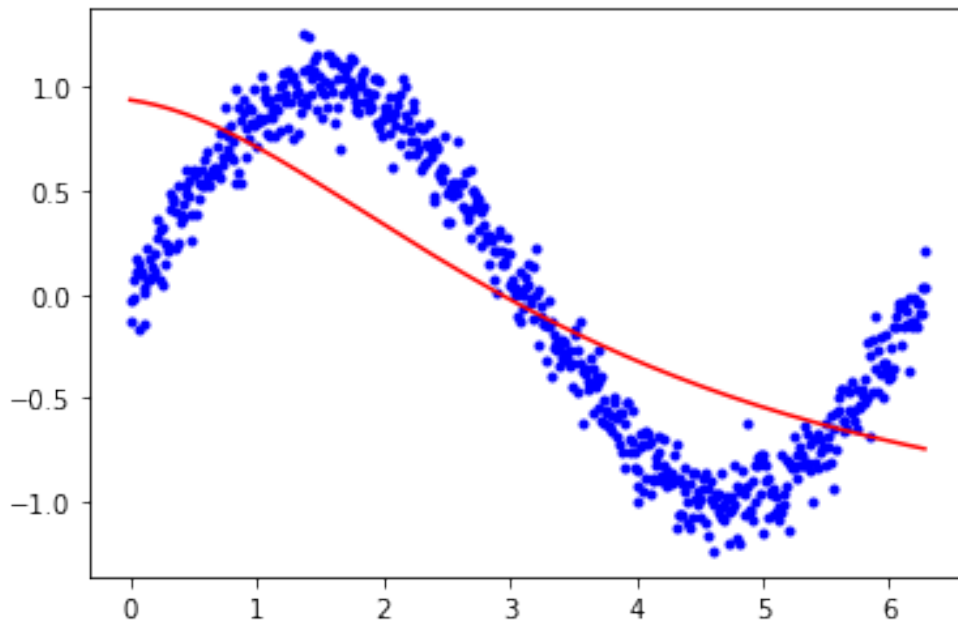


one clearly recognizes the « edges » of the relu function (remind that here, a single hidden layer is used, and the output layer is the identity).

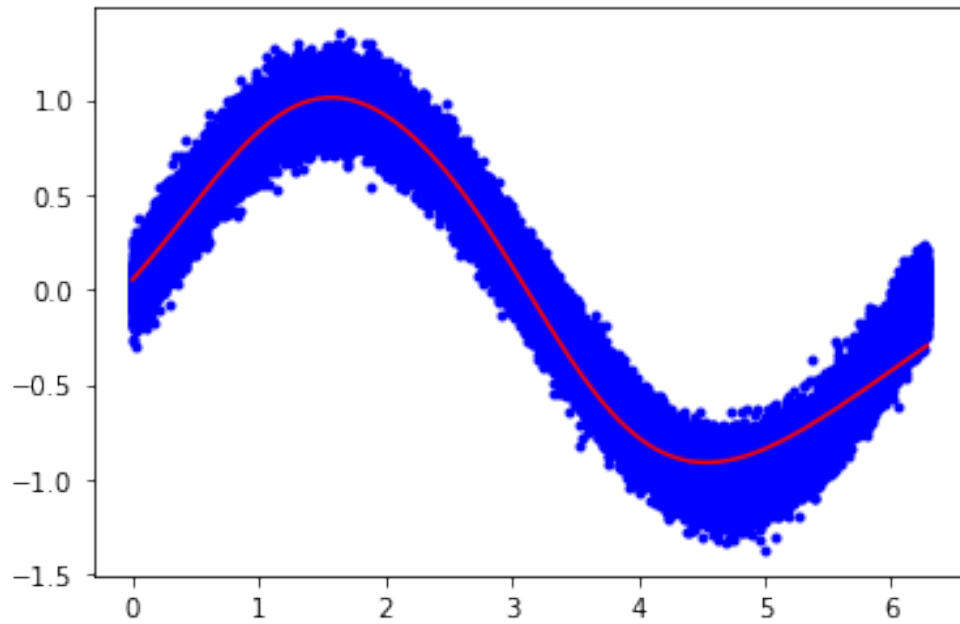
Replacing the activation by « identity », the system remains always linear and is unable to fit the sine function.

2. Using « sgd », and « tanh »

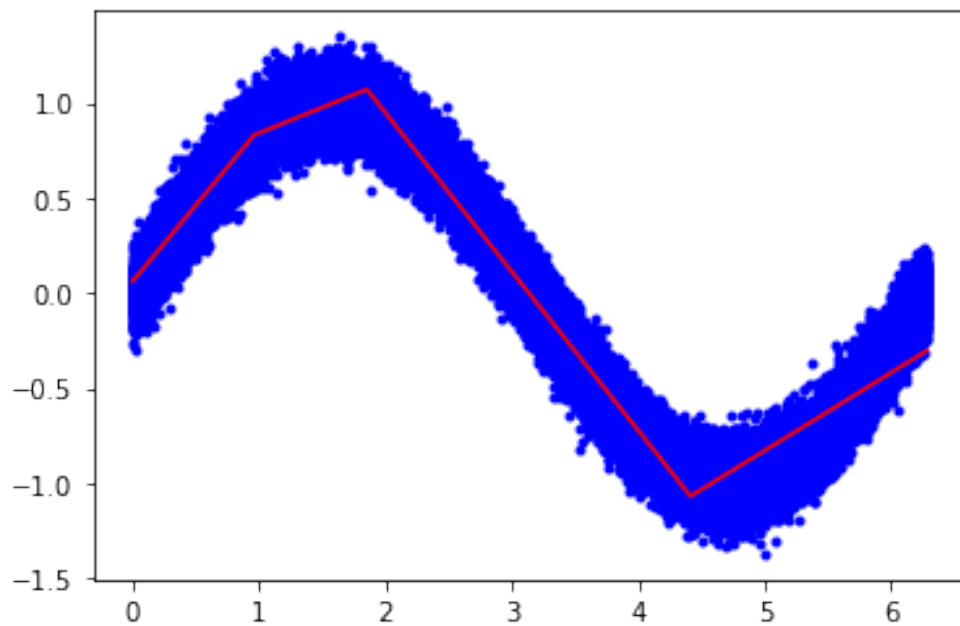
for the single hidden layer case, N-Neurons = 10 , step =.01 :



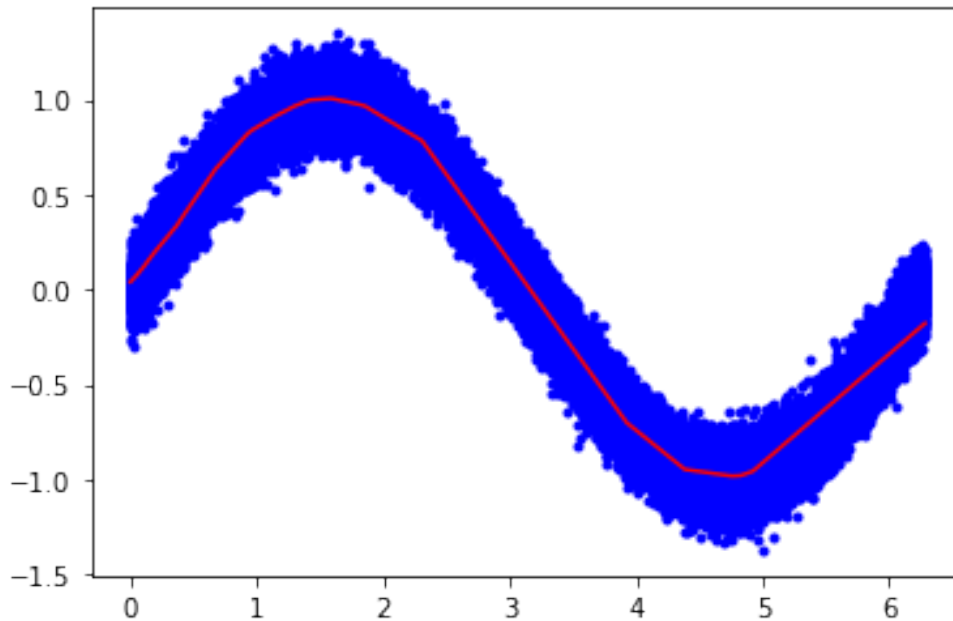
whereas for step=.0001 (i.e. 100 times many more samples), we get a better regression ,at the cost of 100 times more calculations in this case :



For relu activation, the result is much worse, even for $N_{\text{Neurons}} = 10$. Basically, relu activation requires more points for the learning process, and fitting smooth function will necessarily require many hidden layers to approximate with successive locally constant slope 1 approximation.



Setting the parameters to « relu » and 2 hidden layers of size 25 :



In that case

from input to hidden layer 1 = $2 \times N_{\text{neurons}}$ coeffs (2 because x (scalar) + bias)

from hidden layer 1 to hidden layer 2 : $(N_{\text{neurons}} + 1 \text{ (bias)}) \times N_{\text{neurons}}$

from hidden layer 2 to output layer : $N_{\text{neurons}} + 1$

total number of coeffs : $2N + N^2 + N + N + 1 = N_{\text{neurons}}^2 + 4N_{\text{neurons}} + 1$

rq for $N_{\text{neurons}} = 25$, this equals 726! if step is kept to be .01, only 629 data are used...

'sgd' corresponds to perform a random walk with an average decay towards the local minimum. The number of steps to perform will be large. « relu » approximates the data with curves that are made of 2 slopes (one of derivative = 0, and the other one with derivative = $1 \times \text{coeff}$) : approximating a smooth function with such a « rough » activation function will require to have many neurons, and many layers to allow approximations by higher order locally polynomial curves.

To summarize, this increased complexity induces also to use many more samples for solving the optimization problem.