Grenoble INP - Ense3

AI and Autonomous systems

Lab 8 - M2-MARS

MultiLayer Perceptrons, Neural Networks

Author :
Souhaiel BEN SALEM

26 November 2021

Lab Objectives: The objective of this lab is to introduce basic properties of MultiLayer perceptrons, and to get first insights on the role of different choices for hyperparameters (number of layers, number of neurons, choice of activation functions) and optimization algorithm (quasi Newton or stochastic gradient).
We will start with the basics of a single perceptron and move to MLP afterwards.

Introduction:

The perceptron algorithm was invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It is an algorithm for supervised learning of binary classifiers enables neurons to learn and processes elements in the training set one at a time. There are two types of Perceptrons: Single layer and Multilayer:

- Single layer Perceptrons can learn only linearly separable patterns.
- Multilayer Perceptrons or feed-forward neural networks with two or more layers have the greater processing power.

1 Notebook 1: PERCEPTRON INTRODUCTORY EXAMPLES

We will examining the behaviour of the perceptron using as many inputs as coordinates in an input sample (row), on a binary classification problem. The output decision function is the Heaviside step function.

Exercise1:

1. **Show that the predict function defines an separation hyperplan in** $R^{dim(X)}$
   A hyperplane is a higher-dimensional generalization of lines and planes. It is an n−dimensional vector space R^n that is defined to be the set of vectors:

$$u = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{1}$$

   satisfying the equation:

$$a_1x_1 + \; + a_nx_n = b$$

   where $a_1, ..., a_n$ and b are real numbers with at least $a_1, ..., a_n$ non-zero. Our "predict" function computes

$$\text{activation} = \sum_{k=0}^{\dim(X)} w[k].X[k]$$

with $w[0] = bias$ which is the equation of hyperplane in $R^{dim(X)}$ with the bias being added to account for possible affine separation (without the bias term, the hyperplane that w defines would always have to go through the origin).

2. **Express the equation of the separation line as a function of the $\{w[k]\}$ in the case dim(X)=2**
   In the case dim(X)=2, the equation of the separation line is written as follows:

$$\text{activation} = \sum_{k=0}^{\dim(X)} w[k].X[k] = bias + w[1].X[1] + w[2].X2$$

3. **Show in that latter case that setting $w[2]$=0 amounts to define a threshold on the first coordinate**
   Setting $w[2]$=0 means that our activation function becomes

$$\text{activation} = w[0] + w[1].X[1]$$

   which is the equation of a line passing by the bias. Our classification rule becomes:

$$\hat{y} = \begin{cases} +1 \text{ if X[1] = (activation-bias)/ w[1] } \geq 0 \\ -1 \text{ if X[1] = (activation-bias)/ w[1] } < 0 \end{cases}$$

   Which amounts to defining a threshold equals to $(activation - bias)/w[1]$ on the first coordinate.

4. **Propose a set of values for $\{w[0], w[1], w[2]\}$ which defines a good classifier for the data above.**
   A set of $\{w[0], w[1], w[2]\}$ that could define a good classifier of our dataset is $-2, 0.5, 0$ (here we are defining a threshold on the first component). Which give the following decision boundary:

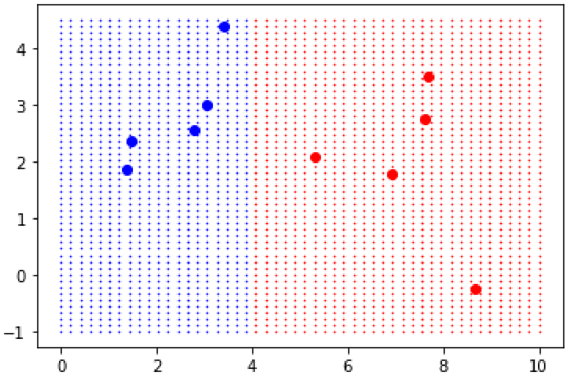


Figure 1: the decision boundary obtained for the chose set of weights

Exercise2:

1. **What is the equation of the boundary?**
   For

$$w = \begin{bmatrix} -4.5 \\ 1 \\ 0 \end{bmatrix} \tag{2}$$

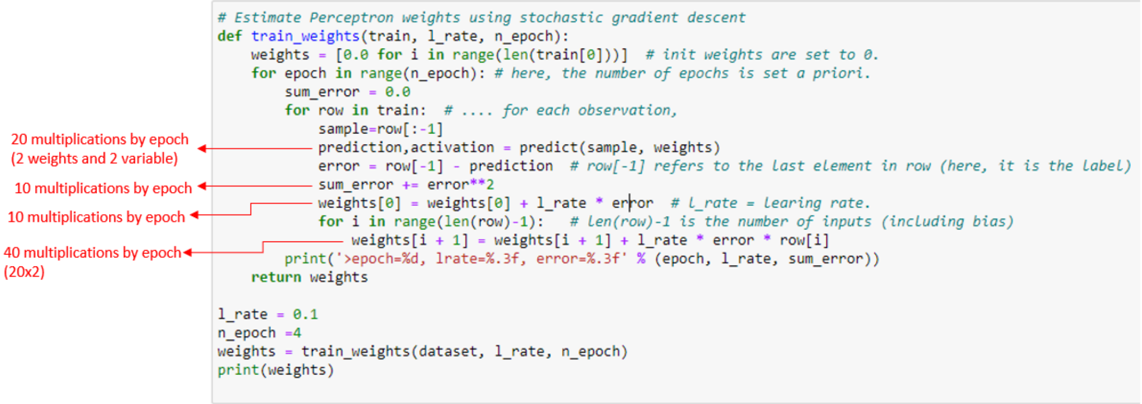
   we get the activation function (decision boundary function): activation $= -4.5 + X[1]$

2. **What is the value of the bias (or intercept)?**
   The intercept (bias) = $w[0]$ = -4.5

3. **Is this solution unique?**
   The solution is not unique, because there are more than one hyperplanes separating two linearly separable classes.For instance, we can fix the intercept and and change the slope pf the hyperplane. Also, the weights vector we gave in question (1.4.) along with the one provides in the notebook define two different separation hyperplanes for the same dataset.

Exercise3:

1. **How many operation (take only multiplicaiton into acocunt) are necessary to complete an epoch in the function train_weights defined above?**
   An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. If we assume that train = dataset, then we would have a total of 80 multiplication operations.
   The figure below illustrates these operations:



```
# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
    weights = [0.0 for i in range(len(train[0]))]  # init weights are set to 0.
    for epoch in range(n_epoch): # here, the number of epochs is set a priori.
        sum_error = 0.0
        for row in train:  # .... for each observation,
            sample=row[:-1]
            prediction,activation = predict(sample, weights)
            error = row[-1] - prediction  # row[-1] refers to the last element in row (here, it is the label)
            sum_error += error**2
            weights[0] = weights[0] + l_rate * error  # l_rate = learing rate.
            for i in range(len(row)-1):   # len(row)-1 is the number of inputs (including bias)
                weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
    return weights

l_rate = 0.1
n_epoch =4
weights = train_weights(dataset, l_rate, n_epoch)
print(weights)
```

20 multiplications by epoch (2 weights and 2 variable)

10 multiplications by epoch

10 multiplications by epoch

40 multiplications by epoch (20x2)

Figure 2: Number of multiplication operation per epoch

Exercise3:

1. **Which of the three representations above is the more informative?**
   The third figure is the most informative because noot only it showa the decision regions but also the vaues of the activation function which can give more insight into the property of each class.

Exercise4:

1. **Using the sklearn reference documentation, identify the role of the parameters ”Shuffle” and ”Validation_fraction”**
   The parameter:

   - **shuffle**: is a boolean (True by default) that indicates Whether or not the training data should be shuffled after each epoch.

   - **validation_fraction**: is a float that represents the proportion of training data to set aside as validation set for early stopping.

2. **Discuss the interest of introducing such parameters**

   - **shuffle**: Shuffling training data, both before training and between epochs, helps prevent overfitting by ensuring that batches are more representative of the entire dataset (in batch gradient descent) and that gradient updates on individual samples are independent of the sample ordering which can prevent error propagation.

   - **validation_fraction**: This parameter is used to set the the proportion of training data to be set aside for validation when an early stopping occurs. Early stopping is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent.

3. **What was the value of parameter ”eta0” in our ”train_weights” code?**
   is equivalent to the learning rate ””l_rate” we used in ”train_weights” code. The parameter ”eta0”

Exercise5:

1. **Do you think that a perceptron (also called single layer perceptron) is a performant classifier for this problem?**
   As discussed earlier, a perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. In this case, the two classes are not linearly separable, so a perceptron would not be able to draw a a correct decision boundary.

1.0.1 Exercise7:

1. **Run the code above a few times : describe the shape of the decision regions that you obtain**
   The decision we obtain seem to separate the two classes quiet accurately, with a nearly circular shap.

2. **Is it in contradiction with the property of (single layer) perceptron qtating that their are linear separators?**
   There is no contradiction in the result obtained because we just mapped the two non linearly separable classes to polar coordinates (where they become linearly separable) and trained our perceptron on that version.

3. **What do you conclude about the choice of the representation space of the analyzed data?**
   The choice of the representation space of the analyzed data is vital for the efficasity of any machine learning algorithm and we saw this in the past labworks where we stated that projecting data into another, more convenient space would make the algorithm more usable nad interpretable (PCA for instance), which is the case for the single perceptron too.

2 Notebook 2: MULTILAYER PERCEPTRON CLASSIFIER

Exercise1 :

1. **Is this 2 clusters problem linearly separable?**
   Yes, this problem is linearly separable as we can visually see that we could fit a plane to separate the two clsses.

2. **how many layers are necessary to separate the two clusters?**
   One single layer consistent of one single perceptron can be used to separate the two clusters shwed above.

3. **Would a Perceptron (similar to the the one studied in N1**$_perceptron.ipynb)beaacceptablesolution$?
   Yes, since the two clusters are linearly separable, one perceptron like the one in the previous notebook can give us an acceptable solution.

Eexercise2:

1. **Would any solution involving a single layer lead to a correct result?**
   Now our data is not linearly separable (i.e a single neuron can not fit a line to separate the two classes). The solution to this problem is to expand beyond the single perceptron architecture by adding an additional layer of units. So to solve this problem and achieve a non-linear separation, we need at least 3 layers (1 input layer, 1 hidden layer and an output layer to combine them both).

2.0.1 Exercise3:

1. **Run the learning process many times with input paramaters hidden_layer_sizes=(3,2). Comment your findings. Propose an interpretation**.
   In this example we are trying to separate two non-linearly separable classes using a MLPN with 2 hidden layers containing 3 units and 2 units respectively.
   We notice that the results we get each run are different, as the MLPN's performance is unstable (sometimes yields a good separation and sometimes does not).

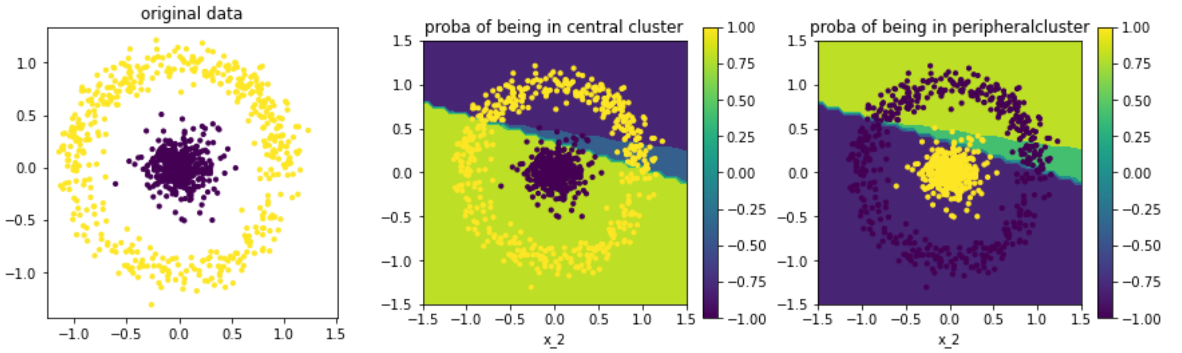


Figure 3: first run (poor result)

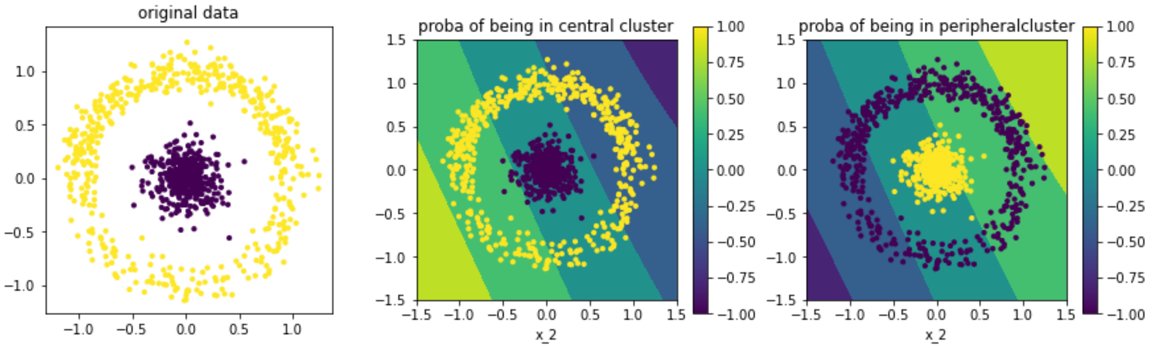


Figure 4: second run (poor result)

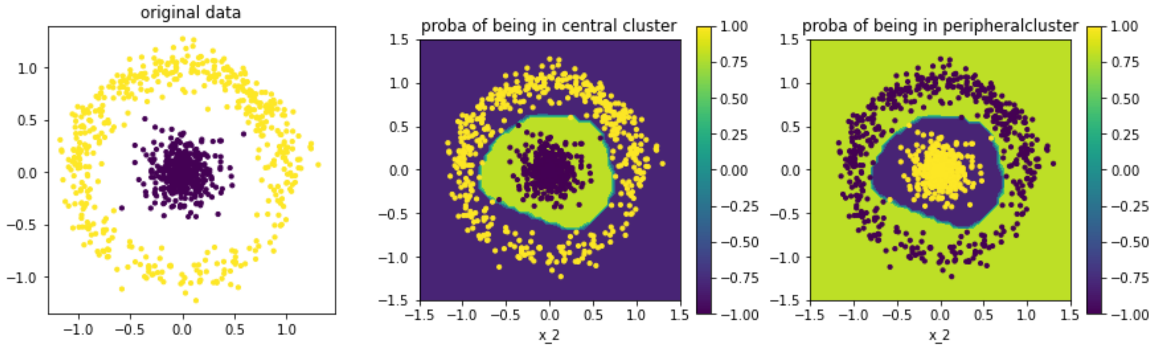


Figure 5: third run (relatively good result))

2. **Increase te number of neurons in the first and second layer and comment your observation. Can you explain your findings?**

We notice that the higher the number of units in the layers, the better the separation boundary shape (more precise) and the better the probability of being in a certain is.
With that being said, we need to keep in mind that adding more units to a single layer means that if the target function had a function at that location in the compositional graph representation of the target function, then we would approximate that better.
Wider layers help train features of the same deep level but in different variations, so we are not increasing the overall complexity of MLPN. We can do that by adding more layers, which essentially allows the MLPN to learn compositional functions that it wasn't able to learn before.
Here are the reuslts obtained for different number of neurons per layer: We also notice that using

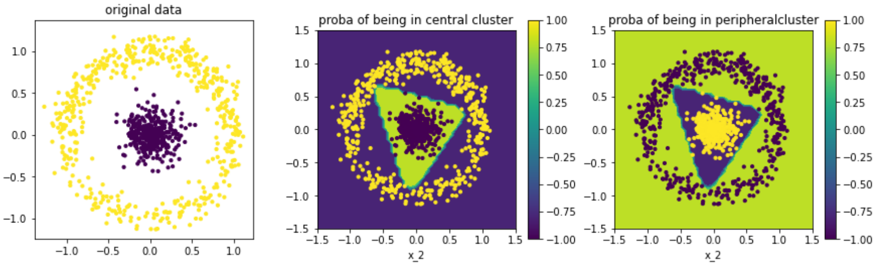


Figure 6: using 3 neurons in each layer

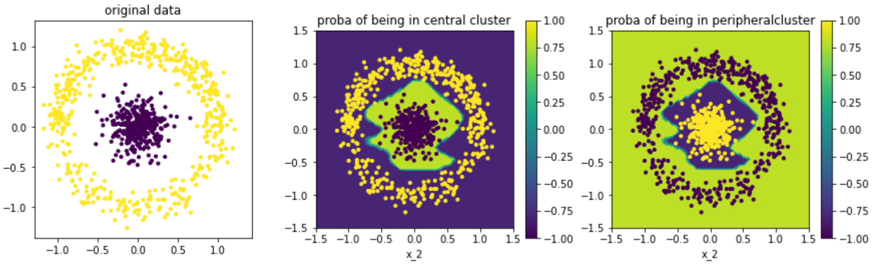


Figure 7: using 10 neurons in each layer

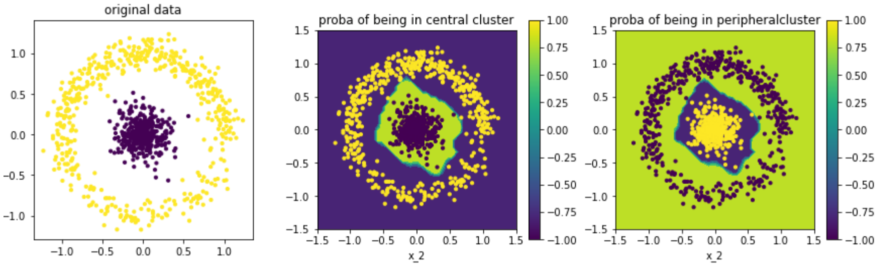


Figure 8: using 20 neurons in each layer

a number of neurons **less or equal to the dimension of our data** (hidden_layer_sizes=(2,10) or hidden_layer_sizes=(20,2) for example) often yields unstable results.

3. **What is different between this method and the method used in N1_Perceptron.ipynb notebook?**
   The method used in N1_Perceptron.ipynb notebook is a change of coordinates where we projected our data points into the polar space and performed linear classification in that space using one perceptron. However, In this example we used a sequence of perceptrons (MLPN) to build a non-linear classifier able to draw complex separation boundaries. The complexity of these boundaries depend on the depth of the MLPN (number of hidden layers) and the number of units per layer.

4. **How would you evaluate the performances of the obatined MLPC?**
   The performance of the MLPC depends of the choice of number of units per hidden layer, if the number is set correctly, the performance of our MLPC is not perfect but acceptable.

3 Notebook 3: Multi-layer Perceptron for regression

Exercise1 : **Using 'tanh' and 'lbfgs' as parameters:**

1. **Change the parameter N**$_{N}eurons(frome.g.2to10)andobservetheresults$
   Using the activation function "tanh" and the solver "lbfgs", we notice that the prediction accurary is better than the model that used "relu" and the "sgd" solver even for much lower number of neurons per layer :

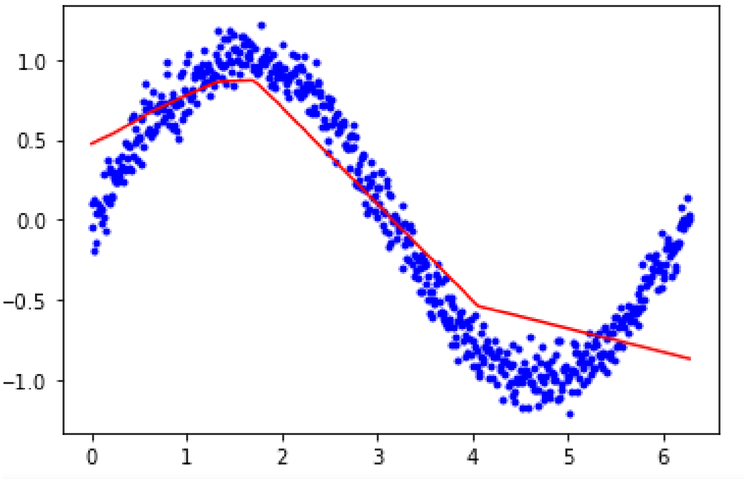


Figure 9: Using "relu" and "sgd" with 25 neurons

This example shows the importance of the choice of the activation function and solder that need to be used for each task.

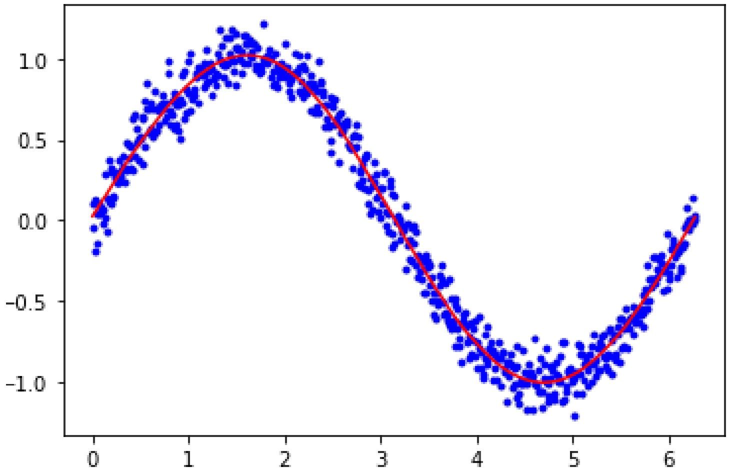


Figure 10: Using ”tanh” and ”lbfgs” with 2 neurons

We also notice that for ”tanh” and ”lbfgs” as parameters, the models begin to show signs of overfitting when increasing the number of neurons:

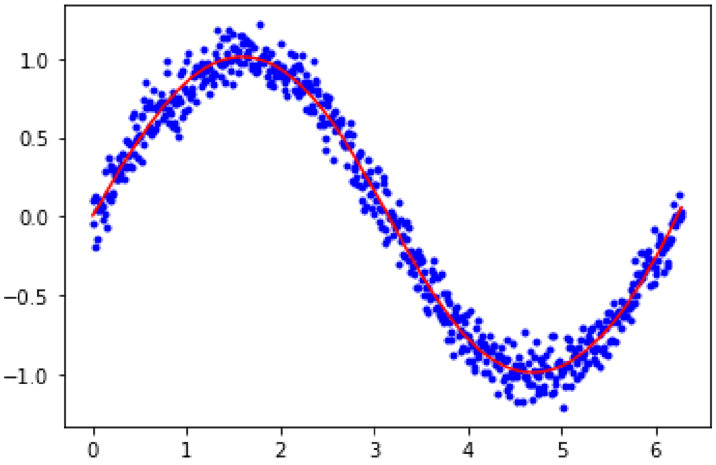


Figure 11: Using ”tanh” and ”lbfgs” with 5 neurons

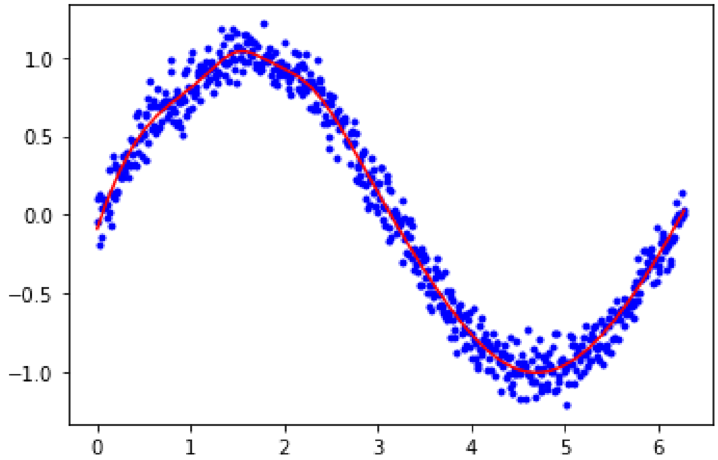


Figure 12: Using ”tanh” and ”lbfgs” with 10 neurons

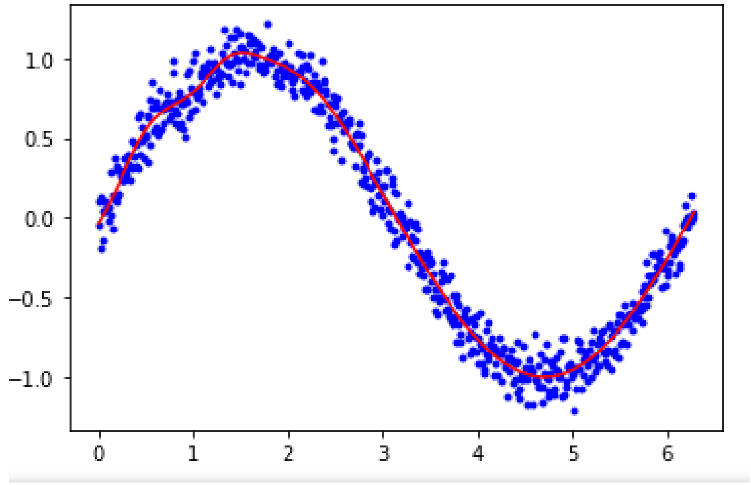


Figure 13: Using ”tanh” and ”lbfgs” with 25 neurons

2. **How many coefficients must be identified for this regressor (as unction of N_Neurons)? (refer to the documentation of scikit-learn MLPRgeressor to understand the structure)**
   We have an MLPR that consists of two hidden layers with N_neurons neurons each, an input layer of 629 input and an output layer of 1 single output (the predicted point). The number of parameters to be learnt is : $629 \times N\_neurons($ weights connecting the input layer to the first hidden layer) $+N\_neurons^2$ (weights connecting the two hidden layers) $+N_neurons($weights connecting the second hidden layer to the output) $2 \times N\_neurons$ (biases in the two hidden layers) + 1 (output). number of parameters = $N\_neurons^2 + 632N\_neurons + 1$

3. **Replace ”tanh” by ”relu” or ”identity”: explain your observations**
   Using the same architecture and setting M_neuron = 25, we get the following result for both the activation functions ”relu” and ”indentity” :

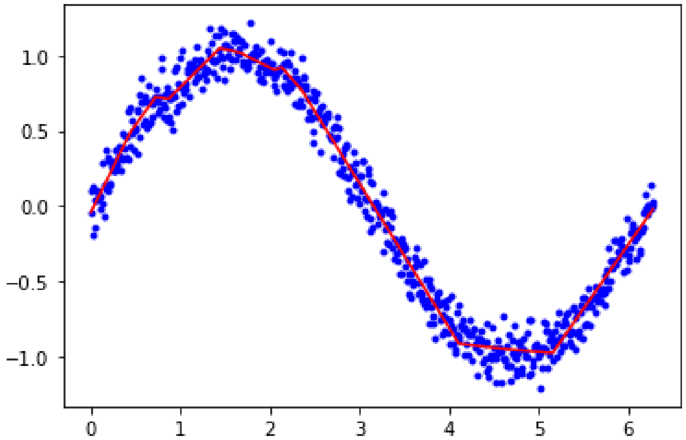


Figure 14: Prediction using ”relu” as an activation function

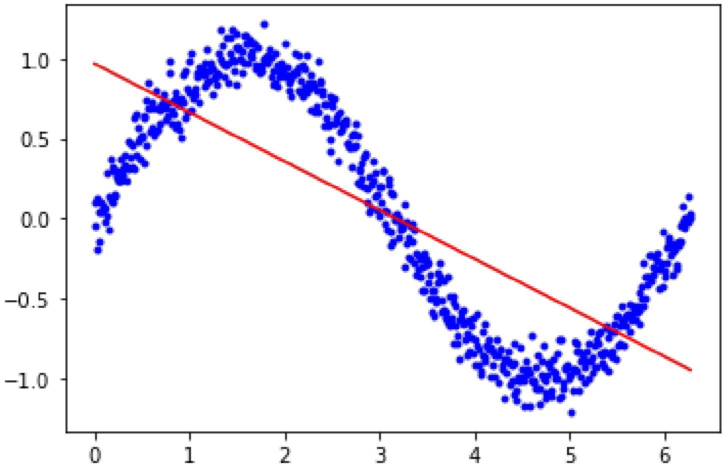


Figure 15: Prediction using ”identity” as an activation function

We notice here that our MLPR performs much better when trained with the non-linear ”relu” function opposed to when trained with the ”identity” function. The result obtained for ”identity” can be explained by the fact that we used a linear activation to train our model to predict a non-linear function which is the sinus function which would naturally lead to inaccurate predictions.

Also, it is worth noting that ”relu” solves the vanishing gradient problem whereas ”identity” does not.

Exercice2: Using ’sgd’ solver

1. **Replace the quasi Newton (lbfgs) solver by a Stochastic Gradient descent (sgd) solver. Comment your observations**
   We notice a performance drom after replacing ”lbfgs” by ”sgd” :

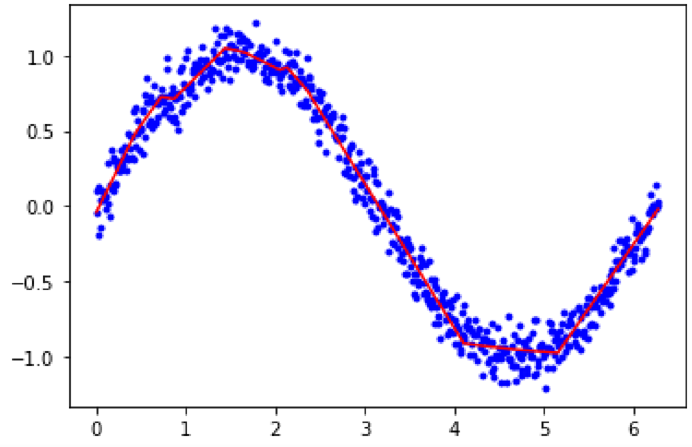


Figure 16: Predictions when using ”lbfgs”

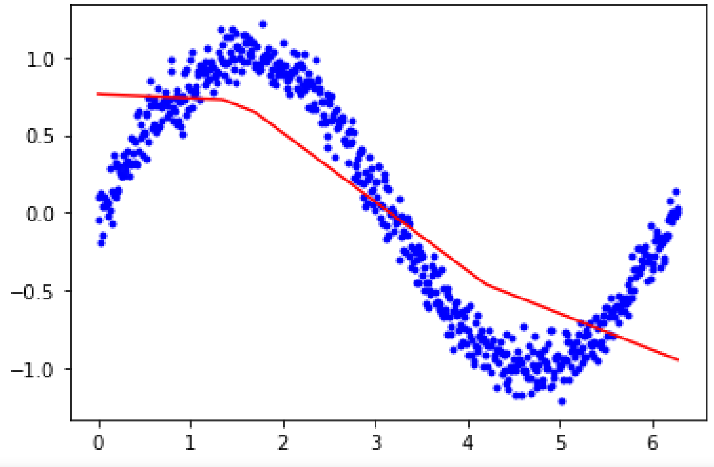


Figure 17: Predictions when using ”sgd”

This problem could be due to ”relu” being Non-differentiable (”sgd”)at zerp, which means The gradients for negative input are zero, which means for activations in that region, the weights are not updated during backpropagation. This can create dead neurons that never get activated.

2. **Replace the parameter ”step” by e.g. 001. and run the code for** **N**$_{N}eurons = 10.Commentyourobservations.$
After setting ”step” = .001, we augmented the nmer of samples considerably which leads to a better better performance of the MLPR :

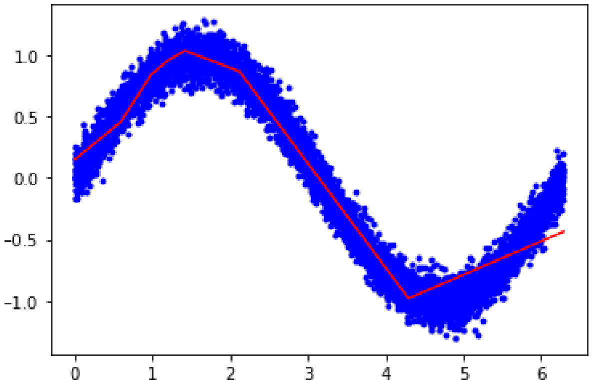


Figure 18: Predictions obtained using 10 neurons after decreasing the ”step”

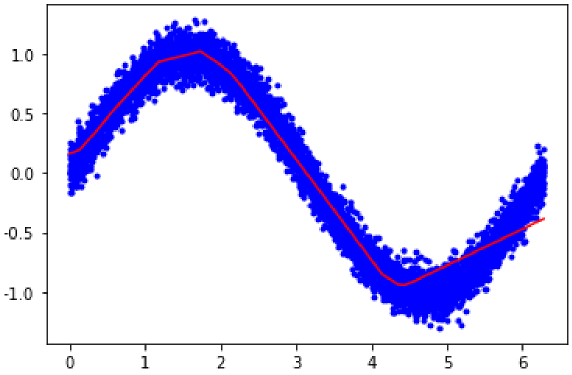


Figure 19: Predictions obtained using 20 neurons after decreasing the ”step”

3. **Set parameter hidden_layer_sizes to (N_Neurons,N_neurons) (2 identical size hidden layers) : how many coefficients need to be identified for this new regressor (in function of N_Neurons)? Set N**$_neuronstoe.g.25.(thismaytakesometimetocompute).Commentyourfindings$
   After setting N_neurons to 25 we notice that the performance of our MLPR (i.e the combination of "relu: and "sgd" needs more layers and epochs to perform better):

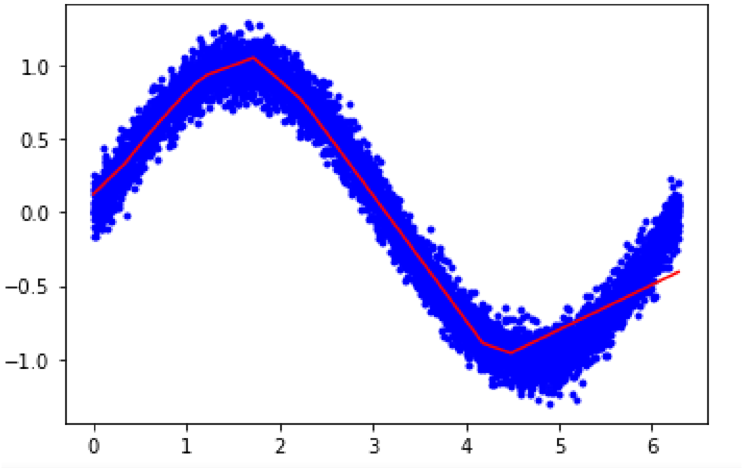


Figure 20: Predictions obtained using 25 neurons and 2 hidden layers

4. **Why does the parameters 'sgd' and 'relu' require more neurons in first layer and at least two layers to give acceptable results?**
   When using ReLU instead of sigmoid or tanh, typicallywe will need to:

   - Decrease your learning rate significantly. This is because ReLU output grows without bound and is much less resistant to high learning rates.
   - Increase number of parameters (i.e. weight) by around 2X or more. This is because of dead relu issue.
   - Increase number of epochs due to much lower learning rate.

Conclusion:

During this lab, we examined the concepts of the perceptron and multilayer perceptron algorithms which are the building blocks of modern day deep learning techniques.
We identified the influence of the architecture (number of hidden layers and neurons per layer), choice of activation function and solver on the performance of the MLPR.