LAB REPORT

MU4IN210 - Robotique et Apprentissage
Regression Labs

Master of Computer Science androide

First year

Academic year 2020 - 2021

Professor : Olivier Sigaud
Students : Vincent Fu
Yuhao Liu

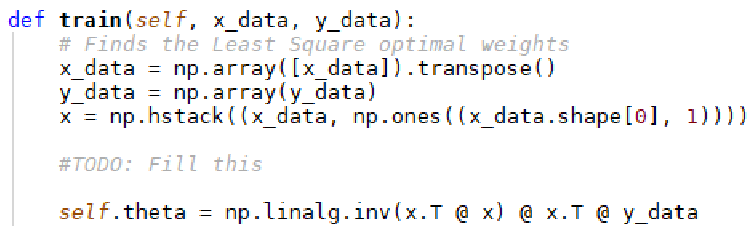
1. Batch Linear Least Squares

Code question 1 :

The function `train` takes measurements `x_data` and `y_data` to perform the linear least squares method and returns the optimal model given by the formula $\theta^* = (\bar{X}^T\bar{X})^{-1}\bar{X}^T y$ with the Gram matrix $\bar{X}$ as

$$\bar{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,d} & 1 \end{pmatrix}$$

where d is the dimension of one sample and N is the number of samples.
The code of `train` is the following:



```
def train(self, x_data, y_data):
    # Finds the Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    #TODO: Fill this

    self.theta = np.linalg.inv(x.T @ x) @ x.T @ y_data
```

For testing those codes, we use the class `SampleGenerator` to generate samples (one point between $[0, 1]$ for one sample). We also compute the sum of the error between the latent function and each data point in order to compare the differences between function performances.
The results are the following:

```
LLS time: 0.00027119999998603816
LLS error: 0.7942569005641469
```

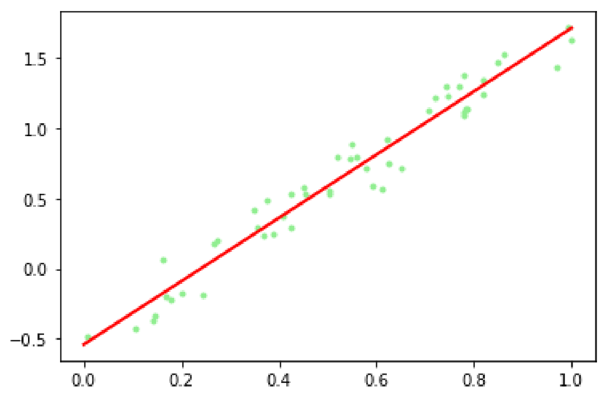


Figure 1: LLS from our function `train`

```
LLS from scipy stats: 0.00045910000000048967
LLS error from scipy stats: 0.7942569005641469
```

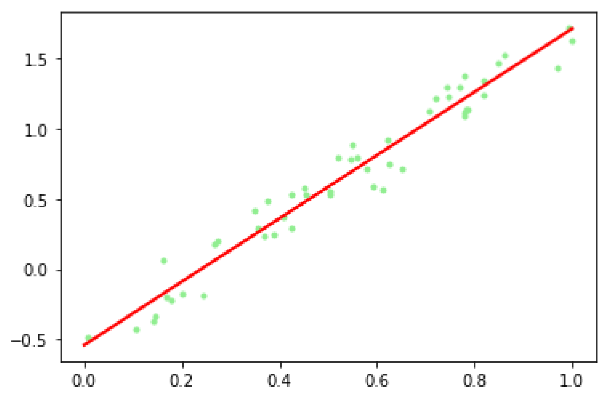


Figure 2: LLS from the function provided by Scipy

We also modify the codes about estimating the duration of a function process by using the function `perf_counter` in order to calculate more accurately when this duration is very short (around 0).
From the two figures, we observe that we have the same error. However, the function `train` is faster than the function `LLS` provided by Scipy but this is not significant because the size of the sample is too small. We also test with a large batch size (for example 10 000) but sometimes the LLS from Scipy is faster. So we can't conclude anything about that.

Code question 2

The function `train_regularized` takes measurements `x_data` and `y_data` to perform the Ridge Regression method and returns the optimal model given by the formula $\theta^* = (\lambda I + \bar{X}^T \bar{X})^{-1} \bar{X}^T y$ where the regularization factor $\lambda \geqslant 0$, I is the identity matrix, and $\bar{X}$ is the same Gram matrix in Code question 1.
The code of the function `train_regularized` is the following:

```
def train_regularized(self, x_data, y_data, coef):
    # Finds the regularized Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    #TODO: Fill this

    self.theta = np.linalg.inv(coef * np.eye(x.shape[1]) + x.T @ x) @ x.T @ y_data
```

The result is the following:

```
regularized LLS : 0.00015529999999941424
regularized LLS error: 1.1629852120697082
```

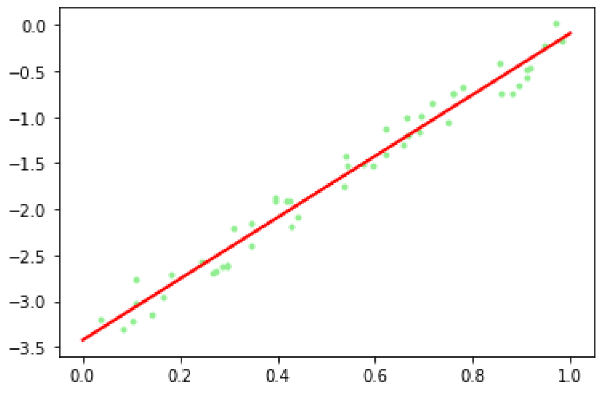


Figure 3: RLLS for 50 data points

We observe that the sum of the error between the latent function and each data point is bigger than the LLS error because this time, we optimize with lower weights in order to avoid potential singularities when we invert matrices.

Study question 3

In order to study how the residuals degrade as we increase the value of `coef`, we plot the evolution of the sum of the error between the latent function and each data points according to the value of `coef`. That's why, we implemented a function called `plot_RRLS_error_evolution` in class `Main`.
The result is the following:

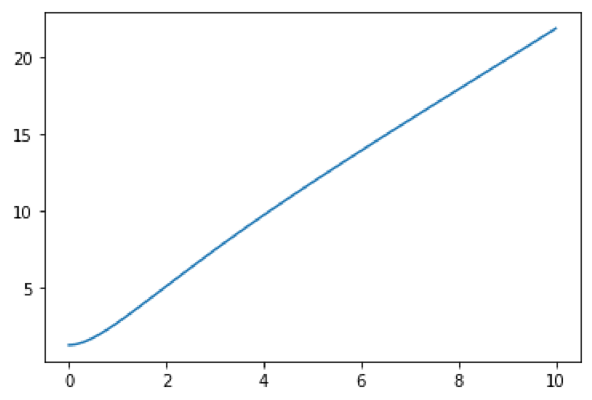


Figure 4: The evolution of the RLLS error depending on `coef` for 50 points

We observe that the residuals degrade linearly as we increase the value of `coef`.

---

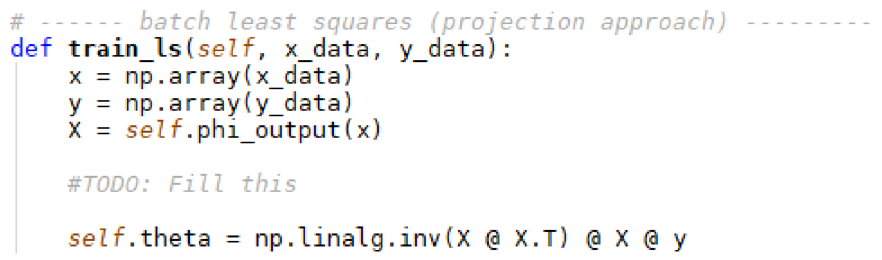
2 Radial Basis Function Networks

Code question 4 :

The function `train_ls` takes measurements `x_data` and `y_data` to perform the least squares method by using radial basis function networks and returns the optimal model given by the formula $\theta^* = (\bar{G}^T\bar{G})^{-1}\bar{G}^T y$ with the Gram matrix $\bar{G}$ as

$$\bar{G} = \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_E(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_E(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_E(x_N) \end{pmatrix}$$

where N is the number of samples, E is the number of radial basis functions with $E < N$ and $(\phi_i)_{1 \leqslant i \leqslant E}$ are radial basis functions (usually they are Gaussian functions).
The code of the function `train_ls` is the following:



```python
# ------ batch least squares (projection approach) ---------
def train_ls(self, x_data, y_data):
    x = np.array(x_data)
    y = np.array(y_data)
    X = self.phi_output(x)

    #TODO: Fill this

    self.theta = np.linalg.inv(X @ X.T) @ X @ y
```

We observe that $X = \bar{G}^T$ because the function `phi_output` returns a vertical vector for one data point.
To find values of the number of the features to lead to good results, we tested different values of `nb_features` from 10 to the batch size and then we got the graphs.
There are some examples of the graphs :

```
RBFN LS time: 0.00020910000012008823
RBFN LS error: 0.026304997549214695
```

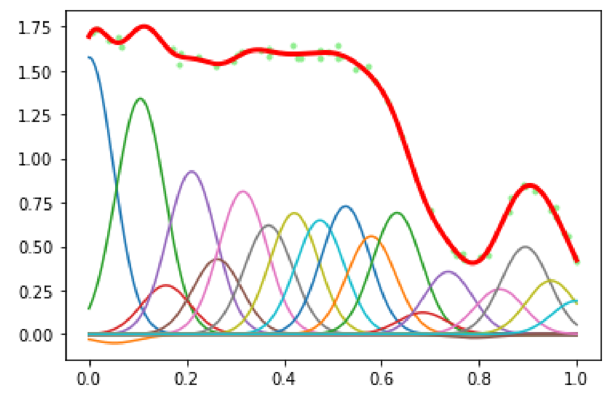


Figure 5: RBFN for 50 data points and 20 features (good results)

```
RBFN LS2 time: 0.0023387999972328544
RBFN LS2 error: 0.011180118453035796
```

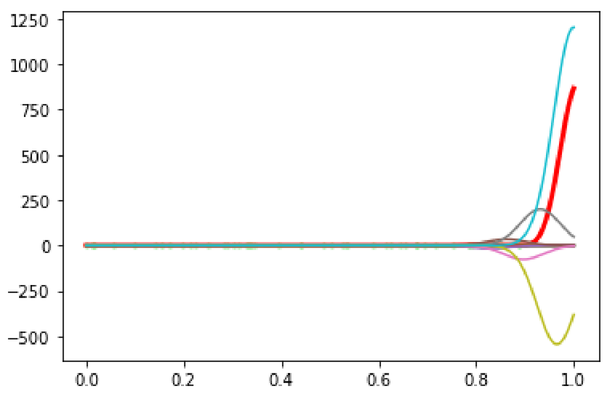
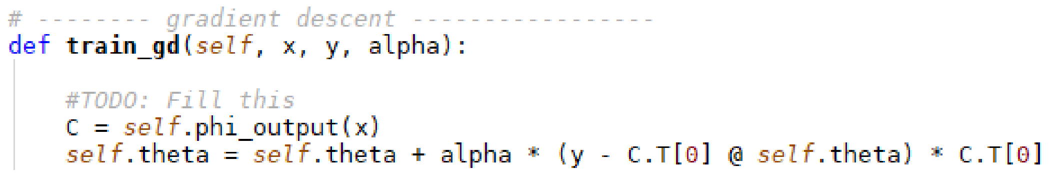


Figure 6: RBFN for 50 data points and 30 features (bad results)

We observe that obtaining good results by selecting an appropriate number of features means that the weights associated to Gaussian functions are not too large compared to our data points (it's the case of figure 5). In the case of figure 6, the weights associated of Gaussian functions are too large so those large features are harmful since they influence the latent function to deviate towards wrong predictions (in the figure 6, we have wrong predictions for $x \in [0.9, 1]$). Therefore, we conclude the number of features leading to good results is between 10 and 20.

Code question 5 :

The function `train_gd` takes measurements `x_data` and `y_data` to perform the vanilla gradient descent method and returns the optimal θ^* by updating incrementally with the update formula at iteration t (for one point): $\theta^{(t+1)} = \theta^{(t)} + \alpha(y^{(t+1)} - \phi(x^{(t+1)})^T\theta^{(t)})\phi(x^{(t+1)})$ where α is learning rate with $\alpha > 0$ and ϕ is a Gaussian feature. The code of the function `train_gd` for one update of θ is the following:



```
# -------- gradient descent -----------------
def train_gd(self, x, y, alpha):

    #TODO: Fill this
    C = self.phi_output(x)
    self.theta = self.theta + alpha * (y - C.T[0] @ self.theta) * C.T[0]
```

In order to find values of `maxIter`, `nb_features` and `alpha` leading to good results, we tested different values and computed the errors between the latent functions and data points. After comparing all the results, we find that `maxIter` $= 1000$, `nb_features` $\in [\![20, 25]\!]$ and `alpha` $\in [0.3, 0.5]$ can lead to good results. However, we analyse that those values depend on each other. It means if we want to get good results, we need to tune correctly `alpha` according to the number of features and to tune correctly `nb_features` according to the value of `maxIter`. That's why `maxIter` $= 2000$, `nb_features` $\in [\![25, 30]\!]$ and `alpha` $\in [0.2, 0.4]$ can also lead to good results.
Examples of those results :

```
RBFN Incr time: 0.0482841999937591
RBFN Incr error: 2.1042286679963227
```

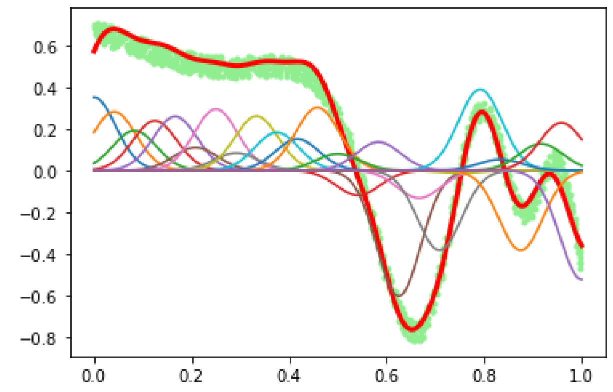


Figure 7: `maxIter` = 1000, `nb_features` = 25, `alpha` = 0.5

```
RBFN Incr time: 0.10530919999473554
RBFN Incr error: 2.5949252751568124
```

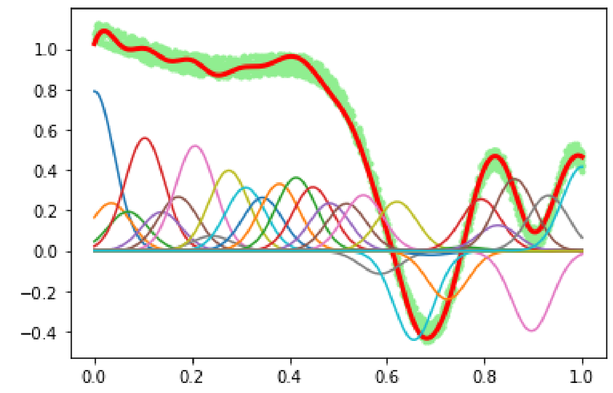


Figure 8: `maxIter` = 2000, `nb_features` = 30, `alpha` = 0.4

The number of features also mustn't be too high compared to the batch size because the higher this number is, the slower the computer computes.

Study question 6 :

In the case of the recursive least square without the Sherman-Morrison formula, we find that `maxIter` = 1000, `nb_features` = 20 lead to a good performance. (There are also some other values which can lead to a good result. This is the same idea as previous question.)
The result is the following:

```
RBFN Incr time: 0.21800669997537625
RBFN Incr error: 0.8657288978900142
```

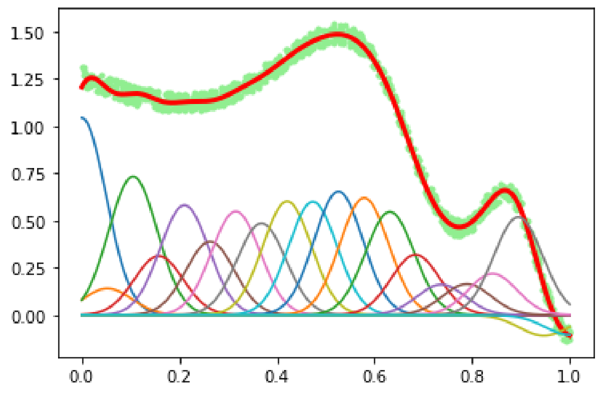


Figure 9: RLS without the Sherman-Morrison formula

In the case of the recursive least square with the Sherman-Morrison formula, we also find that `maxIter` = 1000, `nb_features` = 20 lead to a good performance.
The result is the following:

```
RBFN Incr time: 0.14251030000559695
RBFN Incr error: 1.2539819186889454
```

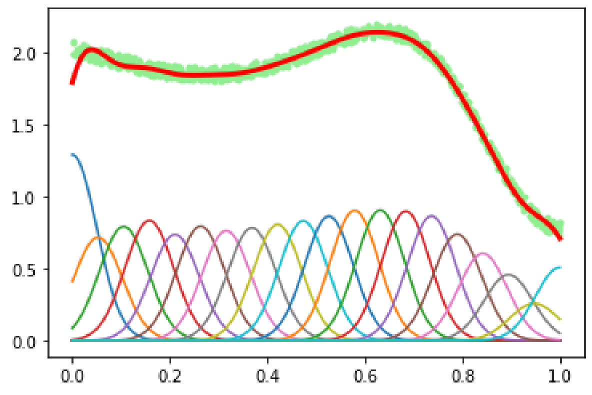
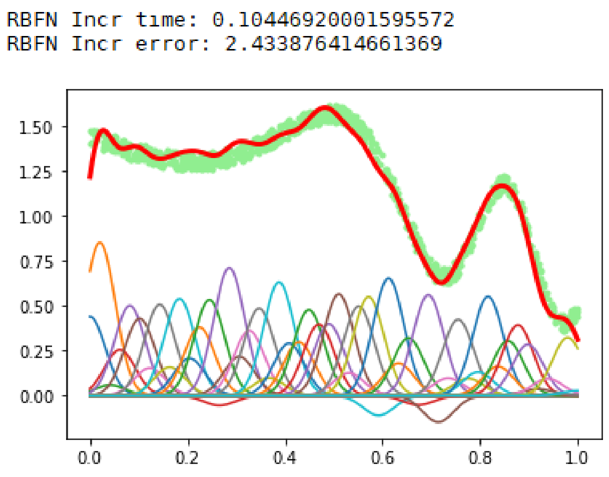


Figure 10: RLS with the Sherman-Morrison formula

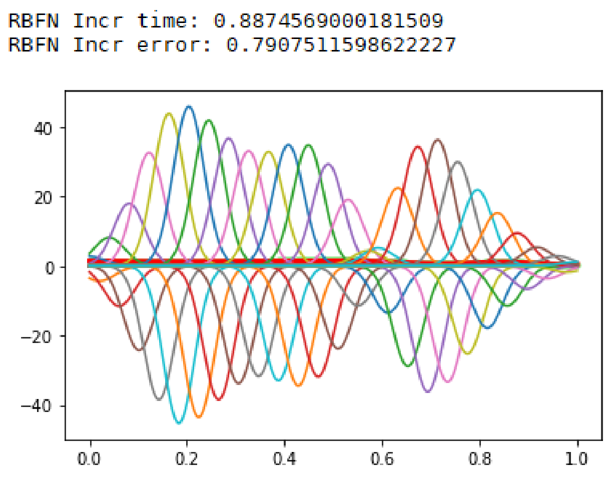
Study question 7 :

In order to compare those 3 methods, we set `maxIter` = 1000, `nb_features` = 50 and `alpha` = 0.5. Those values enable us to find out bad results so we can easily check weaknesses of those 3 methods.
The results are the following:



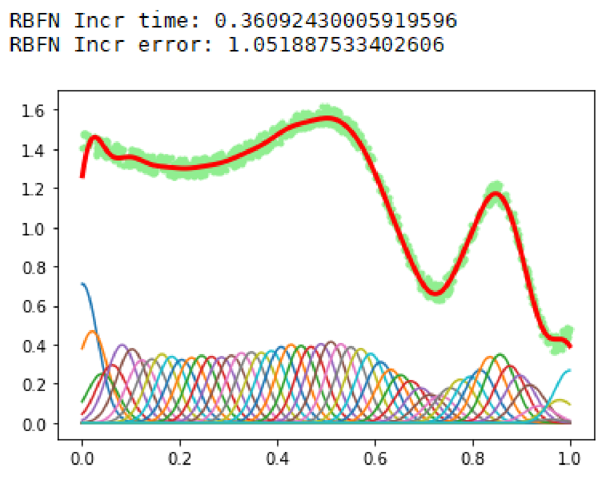
```
RBFN Incr time: 0.10446920001595572
RBFN Incr error: 2.433876414661369
```

Figure 11: RBFN with gradient descent



```
RBFN Incr time: 0.8874569000181509
RBFN Incr error: 0.7907511598622227
```

Figure 12: RBFN with RLS



```
RBFN Incr time: 0.36092430005919596
RBFN Incr error: 1.051887533402606
```

Figure 13: RBFN with RLS and the Sherman-Morrison formula

We observe that the fastest method is still the gradient descent but it has the highest error whereas the most accurate method is the classic recursive least square method. However, this one is over-fitting data points (very large weights associated to some features). Finally, the recursive least square method with the Sherman-Morrison formula is between the two previous methods without over-fitting and manages to get a reasonable error.

Study question 8 :

If we suppose that hyper-parameters are correctly tuned, the main differences between batch and incremental methods are the batch method is more accurate than the incremental method but more expensive in terms of computing when the batch size is large whereas the incremental method is efficient in computing with a large batch but less accurate than the batch method.
So the batch method is stable and expensive whereas the incremental method is unstable but efficient in computing. However, as we saw on the previous questions, we can modify a little bit the incremental method to get a higher performance in order to get more or less stable.
Therefore, if we don't tune correctly the hyper-parameters, it's better to use the incremental methods with recursive least square and the Sherman-Morrison formula than the batch method in order to avoid over-fitting phenomena. Otherwise, when we tune correctly hyper-parameters, if the batch is small, it's better to use the batch method and if the batch is large, it's better to use the incremental method.

---

3. Locally Weighted Least Squares

Code question 9 :

The function `train_lwls` takes measurements `x_data` and `y_data` to perform the locally weighted least squares method and returns the optimal model by combining with each optimal local linear model.
The result is the following :

```
LWR time: 0.03104460000031395
LWR error: 0.053056161648309З
```

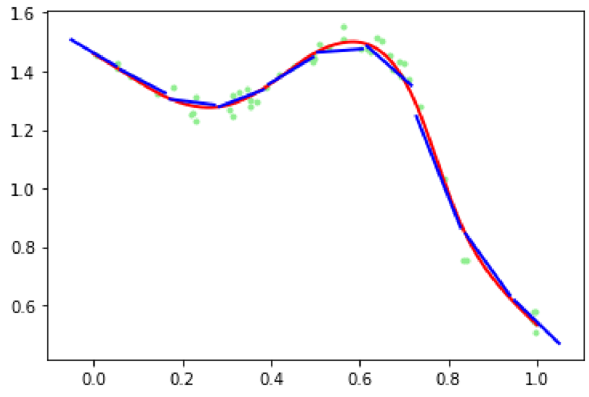


Figure 14: LWLS with 50 data points and 10 local linear models

Study question 10 :

For 50 data points and 10 features, we tested different values of noises and this is an example for `sigma` = 2 :

```
RBFN LS time: 0.00020650000078603625
RBFN LS error: 15.098765658228102
```

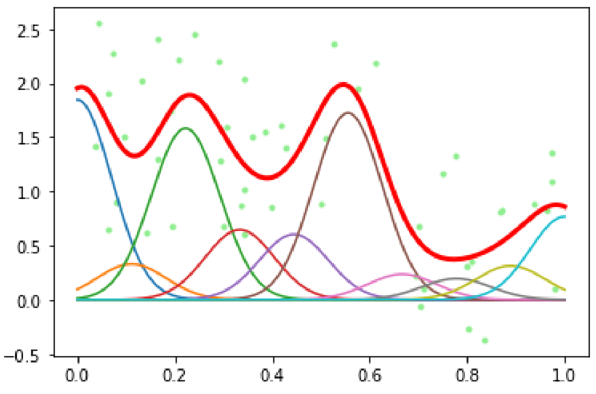


Figure 15: RBFN with the batch method

```
LWR time: 0.04761719999805791
LWR error: 13.784006063663774
```

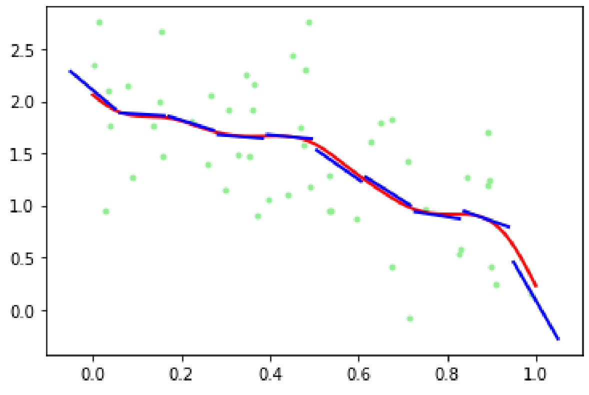
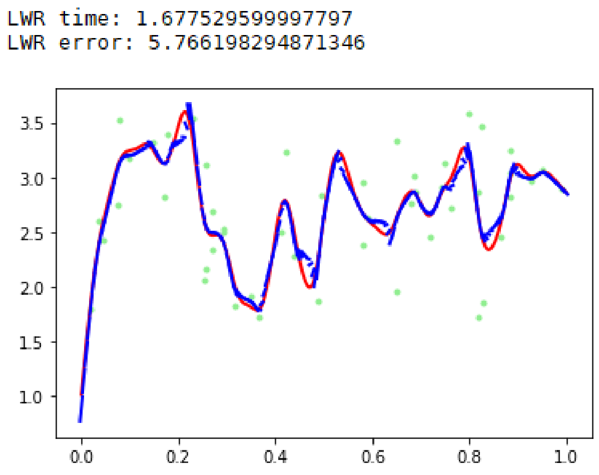


Figure 16: LWLS

We observe that radial basis function networks method is always faster than the locally weighted least square method because the locally weighted least square performs several regressions whereas the radial basis function networks method projects the input space into a feature space to perform only one regression.
That's why when the noises are low, the radial basis function networks method is more accurate than the locally weighted least square method since it needs more features to be more accurate (it's the same idea with discretiza-

tion).
However, when the noises are very high, there is no difference (in terms of accuracy) between those both methods since it's difficult to perform regression. Furthermore, if we increase `nb_features` even if the locally weighed least square method is slower than the radial basis function networks method, in this time, it's more accurate because in the radial basis function networks case, we could get the over-fitting phenomena and the number of features is limited to the batch size whereas the number of features of the locally weighted square method is not limited. The more the number of features is, the more accurate the locally weighted least square method is.
An example of an accurate result with the locally weighted least square method :



```
LWR time: 1.677529599997797
LWR error: 5.766198294871346
```

Figure 17: LWLS with 50 data points and 200 local linear models